

University of Rhode Island

DigitalCommons@URI

Open Access Master's Theses

2015

Training and Source Code Generation for Artificial Neural Networks

Brandon Winrich

University of Rhode Island, bwinrich@my.uri.edu

Follow this and additional works at: <https://digitalcommons.uri.edu/theses>

Recommended Citation

Winrich, Brandon, "Training and Source Code Generation for Artificial Neural Networks" (2015). *Open Access Master's Theses*. Paper 780.
<https://digitalcommons.uri.edu/theses/780>

This Thesis is brought to you for free and open access by DigitalCommons@URI. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons@etal.uri.edu.

TRAINING AND SOURCE CODE GENERATION FOR ARTIFICIAL
NEURAL NETWORKS

BY

BRANDON WINRICH

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2015

MASTER OF SCIENCE THESIS
OF
BRANDON WINRICH

APPROVED:

Thesis Committee:

Major Professor Lutz Hamel

Suzanne Mello-Stark

Resit Sendag

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2015

ABSTRACT

The ideas and technology behind artificial neural networks have advanced considerably since their introduction in 1943 by Warren McCulloch and Walter Pitts. However, the complexity of large networks means that it may not be computationally feasible to retrain a network during the execution of another program, or to store a network in such a form that it can be traversed node by node. The purpose of this project is to design and implement a program that would train an artificial neural network and export source code for it so that the network may be used in other projects.

After discussing some of this history of neural networks, I explain the mathematical principals behind them. Two related training algorithms are discussed: backpropagation and RPROP. I also go into detail about some of the more useful activation functions.

The actual training portion of the project was not self implemented. Instead, a third party external library was used: Encog, developed by Heaton Research. After analyzing how Encog stores the weights of the network, and how the network is trained, I discuss how I used several of the more important classes. There are also details of the slight modifications I needed to make to one of the classes in the library.

The actual implementation of the project consists of five classes, all of which are discussed in the fourth chapter. The program has two inputs by the user (a config file and a training data set), and returns two outputs (a training error report and the source code).

The paper concludes with discussions about additional features that may be implemented in the future. Finally, an example is given, proving that the program works as intended.

ACKNOWLEDGMENTS

I would like to this this opportunity to thank my major professor, Dr. Lutz Hamel, for his assistance and guidance during the duration of this project.

I would also like to thank my friends and family for their support during this process. It took longer than expected, but it worked out well in the end.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGMENTS	iii
TABLE OF CONTENTS	iv
LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER	
1 Background Information	1
1.1 Predecessors to ANNs	1
1.1.1 Perceptron	2
1.2 What is an Artificial Neural Network?	3
1.3 Justification for Study	5
List of References	6
2 Mathematical Elements of Networks	7
2.1 Training	7
2.1.1 Backpropagation	7
2.1.2 Resilient Propagation	9
2.2 Activation Functions	11
2.2.1 Linear	11
2.2.2 Sigmoid	12
2.2.3 Hyperbolic Tangent	12

	Page
2.2.4 Elliott	13
List of References	15
3 Java library Encog	16
3.1 Overview	16
3.2 How Encog Stores Weights	16
3.3 Training	18
3.4 Some Individual Classes	20
3.4.1 TrainingSetUtil	20
3.4.2 BasicNetwork	23
3.4.3 FlatNetwork	24
3.4.4 BasicLayer	25
3.4.5 BasicMLDataSet	25
3.4.6 BasicMLDataPair	26
3.4.7 ActivationFunction	26
4 Implementation	28
4.1 Overview	28
4.2 Assumptions/Design Choices	28
4.3 Inputs	29
4.3.1 Config File	29
4.3.2 Training Data Set	32
4.4 Outputs	32
4.4.1 Training Error Report	33
4.4.2 Source Code	33

	Page
4.5 Individual classes	35
4.5.1 NeuralGenerator	35
4.5.2 LayerInfo	36
4.5.3 OutputWriter	37
4.5.4 OutputWriterTxt	38
4.5.5 OutputWriterJava	39
List of References	41
5 Future work and conclusions	42
5.1 Categorical classification	42
5.2 Additional output formats	42
5.3 Non-command line inputs	43
5.4 Normalization	44
5.5 Conclusions	44
List of References	47
 APPENDIX	
Source Code	49
A.1 NeuralGenerator.java	49
A.2 LayerInfo.java	64
A.3 OutputWriter.java	67
A.4 OutputWriterTxt.java	72
A.5 OutputWriterJava.java	78
A.6 TrainingSetUtil.java (modified)	85
BIBLIOGRAPHY	88

LIST OF TABLES

Table		Page
1	Average number of required epochs	11
2	Elliott vs TANH	15

LIST OF FIGURES

Figure		Page
1	McCulloch-Pitts model of a neuron	1
2	An example of a neural network	4
3	An example of a neural network with bias nodes	5
4	The two sides of a computing unit[1]	7
5	Extended network for the computation of the error function[1] .	8
6	Result of the feed-forward step[1]	9
7	Backpropagation path up to output unit $j[1]$	9
8	Linear activation function	11
9	Sigmoid activation function	12
10	Hyperbolic tangent activation function	13
11	Comparison between Elliott (solid) and sigmoid (dotted) activation functions	14
12	Comparison between Symmetric Elliott (solid) and hyperbolic tangent (dotted) activation functions	14
13	Neural network with labeled weight indexes	16
14	BasicNetwork.getWeight()	17
15	Class hierarchy for training	18
16	Comparison between modified and original code	22
17	Sample from first output file (.txt)	33
18	Sample from first output file (.csv)	33
19	Sample second output file (.txt)	39
20	Sample second output file (.java)	41

Figure		Page
21	test.csv	44
22	output1.csv	45
23	graph of output1.csv	45
24	Results from NeuralGenerator.java	46
25	TestModule.java	46
26	Results from output2.java	46
27	Sample config file	47
28	output2.java	48

CHAPTER 1

Background Information

1.1 Predecessors to ANNs

The history of most neural network research can be traced back to the efforts of Warren McCulloch and Walter Pitts. In their 1943 paper ‘A logical Calculus of Ideas Immanent in Nervous Activity’[1], McCulloch and Pitts introduced the foundation of a neuron, a single piece of the nervous system, which would respond once a certain threshold had been reached. This model of a neuron is still used today.

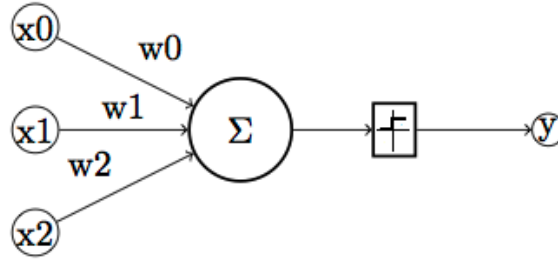


Figure 1: McCulloch-Pitts model of a neuron

The input values to a neuron are all given individual weights. These weighted values are summed together and fed into a threshold function: every value greater than 0 returns a value of 1, and all other values return 0.

In 1949, neuropsychologist Donald Hebb published his book ‘The Organization of Behavior’. Hebb postulated that “When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.” [2]. Hebbian learning influenced research in the field of machine learning, especially in the area of unsupervised learning.

In 1958, Frank Rosenblatt developed the Perceptron. More information on this will be presented in the following subsection.

In 1959, Bernard Widrow and Marcian Hoff developed a working model they called ADALINE (ADaptive LINEar), as well as a more advanced version known as MADALINE (Multiple ADaptive LINEar)[3]. These models were some of the first to be applied to real world problems (such as eliminating echoes on phone lines), and may still be in use today.[4]

Breakthroughs in neural network research declined starting in 1969, when Marvin Minsky and Seymour Papert published their book ‘Perceptrons: an Introduction to Computational Geometry’. In this book, Minsky and Papert claimed Rosenblatt’s perceptron wasn’t as promising as it was originally believed to be. For example, it was unable to correctly classify an XOR function. While this book did introduce some new ideas about neural networks, it also contributed to what was known as ‘the dark age of connectionism’ or an AI winter, as there was a lack of major research for over a decade.

Interest in artificial networks declined, and the focus of the community switched to other models such as support vector machines. [5]

1.1.1 Perceptron

In his 1958 paper, Frank Rosenblatt considered 3 questions[6]:

1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

The first question wasn't addressed as he believed it "is in the province of sensory physiology, and is the only one for which appreciable understanding has been achieved." The other two questions became the basis of his concept of a perceptron (which he compared to the retina in an eye).

A perceptron functions as a single neuron, accepting weighted inputs and an unweighted bias, the output of which is passed to a transfer function. This step function evaluates to 1 if the value is positive, and either 0 or -1 if the value is negative (the exact value may vary depending on the model). After iterations with a learning algorithm, the perceptron calculates a decision surface to classify a data set into two categories.

The perceptron learning algorithm is as follows[7]:

1. Initialize the weights and threshold to small random numbers.
2. Present a pattern vector $(x_1, x_2, \dots, x_n)^t$ and evaluate the output of the neuron.
3. Update the weights according to $w_j(t+1) = w_j(t) + \eta(d - y)x_j$, where d is the desired output, t is the iteration number, and η ($0.0 < \eta < 1.0$) is the gain (step size).

Steps two and three are repeated until the data set has been properly classified. Unfortunately, due to the nature of the perceptron, it will only work with data that is linearly separable.

1.2 What is an Artificial Neural Network?

An artificial neural network (sometimes referred to as an ANN, or just a neural network) is a machine learning model inspired by biological neural networks (such as the central nervous system).

Neural networks fall under the supervised learning paradigm. In supervised learning, the network is presented with pairs of data, input and output. The goal is to be able to map the input to the output, training in a way that minimizes the error between the actual output and the desired output. More information may be found in section 2.1.

Each piece of the network is known as a neuron. Neural networks still use the McCulloch-Pitts model of a neuron (see figure 1). The inputs into a node are the values from the previous layer (or the input values, if the layer in question is the input layer). Each value is multiplied by the associated weight, and then those products are summed together ($\sum w_i x_i$). Rather than being fed into a threshold function, an activation function is used. This allows for a wider range of potential output values, instead of just 0 or 1. The output value of the neuron can be used as the input into the next layer, or as the output for the network.

Neural networks consist of at least three layers: an input layer, at least one hidden layer, and an output layer:

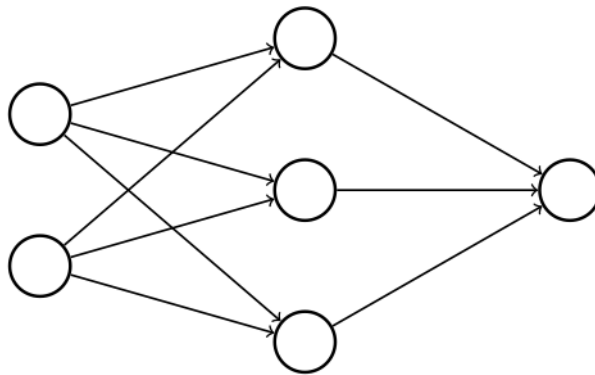


Figure 2: An example of a neural network

It is also possible to have bias nodes. These nodes hold a constant value (often +1), and act only as an input to another neuron (they do not have any inputs or activation functions associated with themselves).

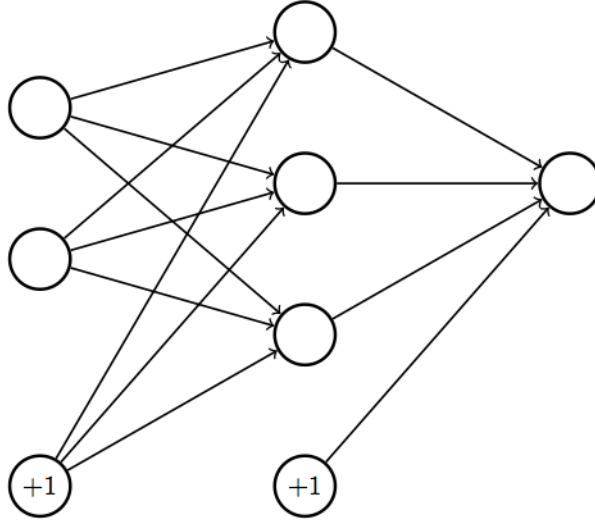


Figure 3: An example of a neural network with bias nodes

1.3 Justification for Study

My main interest in machine learning pertains to the realm of video games. Artificial intelligence is an important aspect of every game (except those which are exclusively multiplayer, with no computer-controlled agents). 5-60% of the CPU is utilized by AI-related processes, and this number has been known climb as high as 100% for turn-based strategy games[8]. While some modern games utilize machine learning, most of this is done before the game is published (rather than the training occurring during runtime). According to Charles and McGlinchey, “online learning means that the AI learns (or continues to learn) whilst the end product is being used, and the AI in games is able to adapt to the style of play of the user. Online learning is a much more difficult prospect because it is a real-time process and many of the commonly used algorithms for learning are therefore not suitable.”[9]

The project that I have completed focuses on generating the source code for an artificial neural network, which is directly applicable to the field of gaming. With

the actual training occurring during the development phase, it makes sense to have a program that can create the network, separate from the rest of the project. The source code that it outputs then allows the network to be used within the context of a game. The other benefit of such a program is that it allows the neural network to be used without having to maintain the structure of the network. Reducing the results of the network down to mathematical formulas results in faster computation times than having to walk through the nodes of a network (as stored in multiple classes or data structures). The results of this project have been tested in a Quake II environment.

List of References

- [1] W. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943. [Online]. Available: <http://dx.doi.org/10.1007/BF02478259>
- [2] D. Hebb, “The organization of behavior; a neuropsychological theory.” 1949.
- [3] B. Widrow, M. E. Hoff, *et al.*, “Adaptive switching circuits.” 1960.
- [4] C. Clabaugh, D. Myszewski, and J. Pang. “History: The 1940’s to the 1970’s.” [Online]. Available: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html>
- [5] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: <http://dx.doi.org/10.1007/BF00994018>
- [6] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*, vol. 65, no. 6, p. 386, 1958.
- [7] A. K. Jain, J. Mao, and K. Mohiuddin, “Artificial neural networks: A tutorial,” 1996.
- [8] L. Galway, D. Charles, and M. Black, “Machine learning in digital games: a survey,” *Artificial Intelligence Review*, vol. 29, no. 2, pp. 123–161, 2008.
- [9] D. Charles and S. McGlinchey, “The past, present and future of artificial neural networks in digital games,” *Proceedings of the 5th international conference on computer games: artificial intelligence, design and education*, pp. 163–169, 2004.

CHAPTER 2

Mathematical Elements of Networks

2.1 Training

2.1.1 Backpropagation

The concept of the backpropagation algorithm was first developed by Paul Werbos in his 1974 PhD thesis, ‘Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences’.

Much of the math in this section comes from Raúl Rojas’ book ‘Neural Networks - A Systematic Introduction’[1]

The backpropagation algorithm works by using the method of gradient decent. In order to do this, it needs to use an activation function which is differentiable. This is a change from the perceptron, which used a step function. One of the more popular activations functions is the sigmoid function. This and other alternatives will be explored in section 2.2.

In order to make the calculations easier, each node is considered in two separate parts. Rojas calls this a *B-diagram* (or backpropagation diagram). As seen in figure 4, the right side calculates the output from the activation function, while the left side computes the derivative.

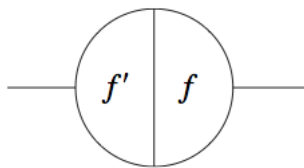


Figure 4: The two sides of a computing unit[1]

Rather than calculating the error function separately, the neural network is extended with an additional layer used to calculate the error internally (as seen

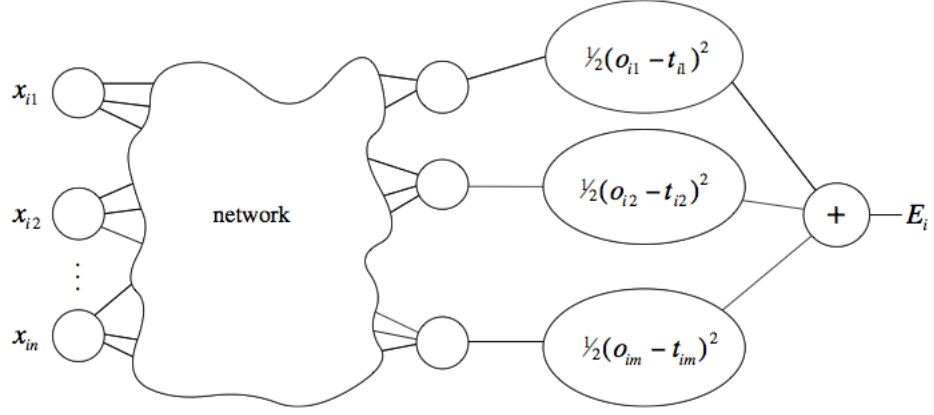


Figure 5: Extended network for the computation of the error function[1]

in figure 5). The equation for the error function is $E = \frac{1}{2} \sum_{i=1}^p ||\mathbf{o}_i - \mathbf{t}_i||^2$, where \mathbf{o}_i is the output value from node i , and \mathbf{t}_i is the target value. Keeping in mind the separation of the nodes as previously mentioned, the derivative calculated in the left portion will be $(\mathbf{o}_i - \mathbf{t}_i)$.

The backpropagation algorithm consists of four steps:

1. Feed-forward computation
2. Backpropagation to output layer
3. Backpropagation to hidden layer(s)
4. Weight updating

In the first step, the algorithm is processed in a straight forward manner, with the output from one node being used as the input to the next node, as seen in figure 6.

Generally speaking, backpropagation retraces through the network in reverse. Since the network is being run backwards, we evaluate using the left side of the node (the derivative). Instead of outputs being used as the the inputs to the next node, outputs from a node are multiplied by the output of previous nodes.

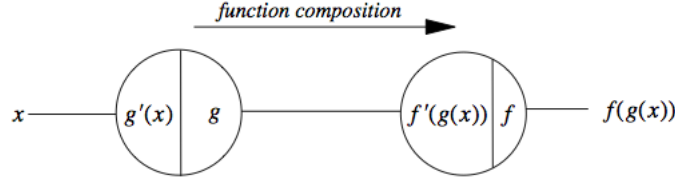


Figure 6: Result of the feed-forward step[1]

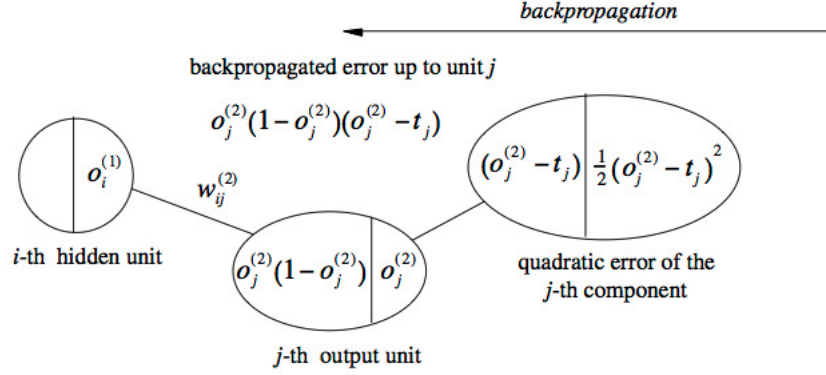


Figure 7: Backpropagation path up to output unit j [1]

We extended the network to calculate the error function, so for the output layer we use that derivative as an input, as seen in figure 7.

Backpropagation for the hidden layer(s) acts in the same way, using the values from the output layer as its input.

The final step is weight updating. The formula for updating the weight w_{ij} (the weight between node i and node j) is $\Delta w_{ij} = -\gamma o_i \delta_j$, where γ is the learning rate, o_i is the output from node i , and δ_j is the error from node j .

A possible variation is the inclusion of a momentum variable η . This can help make the learning rate more stable: $\Delta w_{ij}(t) = -\gamma o_i \delta_j + \eta \Delta w_{ij}(t-1)$

2.1.2 Resilient Propagation

A promising alternative to backpropagation is resilient propagation (often referred to as RPROP), originally proposed by Martin Riedmiller and Heinrich

Braun in 1992. Instead of updating the weights based on how large the partial derivative of the error function is, the weights are updated based on whether the partial derivative is positive or negative.

First, the change for each weight is updated based on if the derivative has changed signs. If such a change has occurred, that means the last update was too large, and the algorithm has passed over a local minimum. To counter this, the update value will be decreased. If the sign stays the same, then the update value is increased.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}^{(t-1)}, & \text{if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} * \frac{\delta E}{\delta w_{ij}}^{(t)} > 0 \\ \eta^- * \Delta_{ij}^{(t-1)}, & \text{if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} * \frac{\delta E}{\delta w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)}, & \text{else} \end{cases} \quad \text{where } 0 < \eta^- < 1 < \eta^+.$$

Typically, η^+ is assigned a value of 1.2, and η^- is assigned a value of 0.5.

Once the update value is determined, the sign of the current partial derivative is considered. In order to bring the error closer to 0, the weight is decreased if the partial derivative is positive, and increased if it is negative.

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)}, & \text{if } \frac{\delta E}{\delta w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^{(t)}, & \text{if } \frac{\delta E}{\delta w_{ij}}^{(t)} < 0 \\ 0, & \text{else} \end{cases}$$

At the end of each epoch, all of the weights are updated:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$$

The exception to this rule is if the partial derivative has changed signs, then the previous weight change is reversed. According to Reidmiller and Braun, “due to that ‘backtracking’ weight-step, the derivative is supposed to change its sign once again in the following step. In order to avoid a double punishment of the update-value, there should be no adaptation of the update-value in the succeeding step.”

$$\Delta w_{ij}^{(t)} = -\Delta w_{ij}^{(t-1)}, \text{ if } \frac{\delta E}{\delta w_{ij}}^{(t-1)} * \frac{\delta E}{\delta w_{ij}}^{(t)} < 0$$

In most cases, the update value is limited to a specific range, with an upper limit of $\Delta_{max} = 50.0$ and a lower limit of $\Delta_{min} = 1e^{-6}$.

Reidmiller and Braun provide tested RPROP against several other popular

algorithms: backpropagation (BP), SuperSAB (SSAB), and Quickprop (QP)[2]:

Problem	10-5-10	12-2-12	9 Men's Morris	Figure Rec.
BP (best)	121	>15000	98	151
SSAB (best)	55	534	34	41
QP (best)	21	405	34	28
RPROP (std)	30	367	30	29
RPROP (best)	19	322	23	28

Table 1: Average number of required epochs

2.2 Activation Functions

2.2.1 Linear

One of the simpler activation functions is the linear function:

$$f(x) = x$$

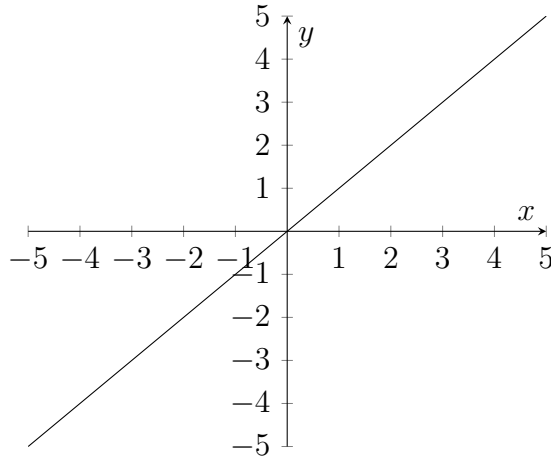


Figure 8: Linear activation function

This activation is very simple, and isn't used very often. The input is directly transferred to the output without being modified at all. Therefore, the output range is \mathbb{R} . The derivative of this activation function is $f'(x) = 1$.

A variation on this is the ramp activation function. This function has an upper and lower threshold, where all values below the lower threshold are assigned a certain value and all values above the upper threshold are assigned a different value

(0 and 1 are common). The result is something similar to the step function used in the perceptron, but with a linear portion in the middle instead of a disjuncture.

2.2.2 Sigmoid

One of the more common activation functions is the sigmoid function. A sigmoid function maintains a shape similar to the step function used in perceptrons (with horizontal asymptotes at 0 and 1). However, the smooth curve of the sigmoid means that it is a differentiable function, so it can be used in backpropagation (which requires an activation function to have a derivative).

$$f(x) = \frac{1}{1+e^{-x}}$$

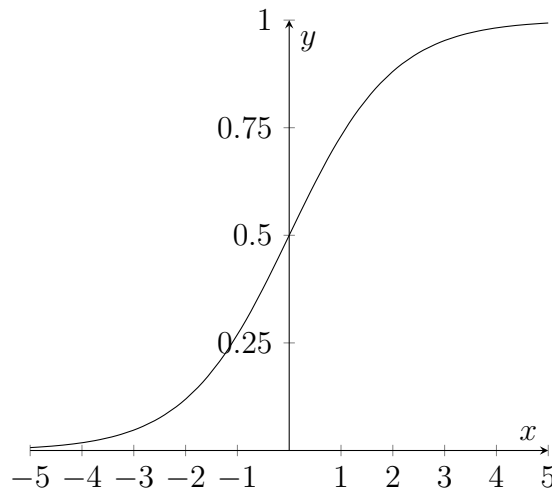


Figure 9: Sigmoid activation function

The output range of this activation function is 0 to 1. The derivative of this activation function is $f'(x) = f(x) * (1 - f(x))$.

2.2.3 Hyperbolic Tangent

The hyperbolic tangent function has a similar shape to the sigmoid function. However, its lower horizontal asymptote is at -1 instead of 0. This may be more useful with some data sets, where use of a sigmoid activation function does not

produce any negative numbers.

$$f(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

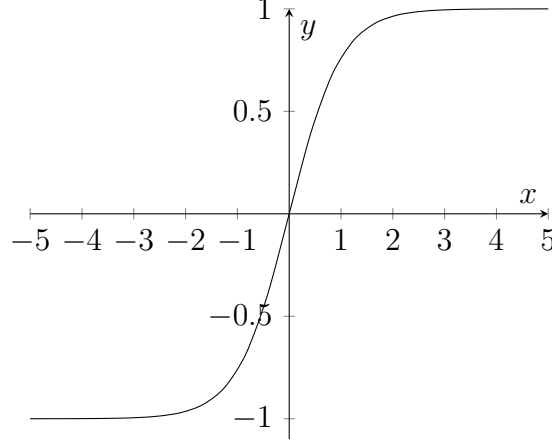


Figure 10: Hyperbolic tangent activation function

The output range of this activation function is -1 to 1. The derivative of this activation function is $f'(x) = 1 - f(x) * f(x)$.

2.2.4 Elliott

Elliott activation functions were originally proposed by David L. Elliott in 1993 as more computationally effective alternatives to the sigmoid and hyperbolic tangent activation functions.[3]

Encog provides two such activation functions: Elliott and Symmetric Elliott. In all of the cases below, s is the slope, which has a default value of 1 (although this can be changed).

The Elliott activation function serves as an alternative to the sigmoid activation function:

$$f(x) = \frac{0.5(x*s)}{1+|x*s|} + 0.5$$

Just as the sigmoid activation function, this produces an output range of 0 to 1. The derivative of this activation function is $f'(x) = \frac{s}{2*(1+|x*s|)^2}$

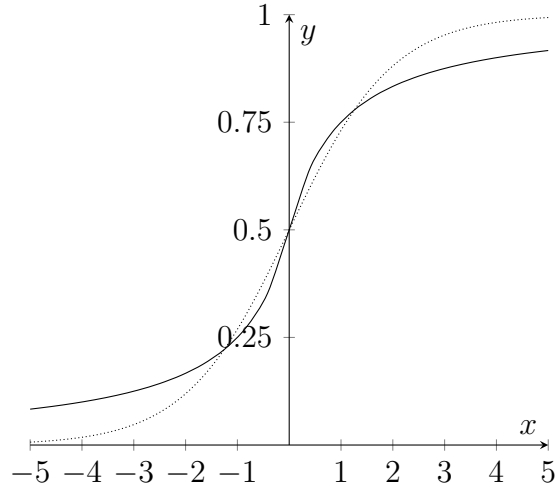


Figure 11: Comparison between Elliott (solid) and sigmoid (dotted) activation functions

The Symmetric Elliott activation functions serves as an alternative to the hyperbolic tangent activation function:

$$f(x) = \frac{x*s}{1+|x*s|}$$

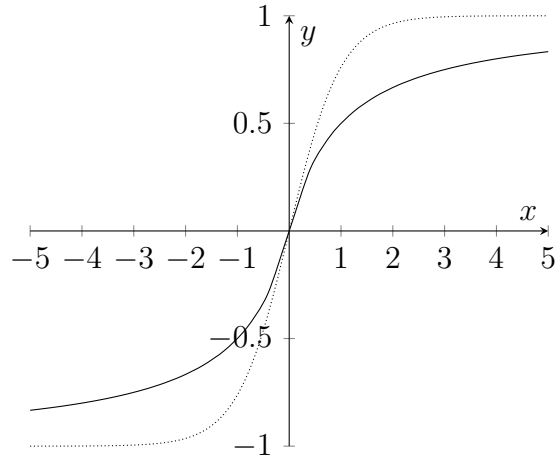


Figure 12: Comparison between Symmetric Elliott (solid) and hyperbolic tangent (dotted) activation functions

Just as the hyperbolic tangent activation function, this produces an output range of -1 to 1. The derivative of this activation function is $f'(x) = \frac{s}{(1+|x*s|)^2}$

Heaton Research (the company that makes the Encog library) provided some interesting statistics on the efficiency of this activation function[4]:

Activation Function	Total Training Time	Avg Iterations Needed
TANH	6,168ms	474
ElliottSymmetric	2,928ms	557

Table 2: Elliott vs TANH

While the Symmetric Elliot required more iterations of training in order to reach the desired error, the time it took for each training iteration was much less than the hyperbolic tangent, resulting in the network being trained in effectively half the time. Although computational power has increased considerably since David L. Elliott first proposed these activation functions (earlier versions of Encog approximated the value of the hyperbolic tangent because it was faster than Java’s built-in TANH function), they can still be useful for training large networks and/or data sets.

According to the javadoc comments for the two classes, these activation functions approach their horizontal asymptotes more slowly than their traditional counterparts, so they “might be more suitable to classification tasks than predictions tasks”.

List of References

- [1] R. Rojas, “The backpropagation algorithm,” in *Neural Networks*. Springer, 1996, pp. 149–182.
- [2] M. Riedmiller and H. Braun, “A direct adaptive method for faster backpropagation learning: The rprop algorithm,” in *Neural Networks, 1993., IEEE International Conference on*. IEEE, 1993, pp. 586–591.
- [3] D. L. Elliott, “A better activation function for artificial neural networks,” 1993.
- [4] J. Heaton. “Elliott activation function.” September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Elliott_Activation_Function

CHAPTER 3

Java library Encog

3.1 Overview

Encog is an external machine learning library. Originally released in 2008, Encog is developed by Heaton Research (run by Jeff Heaton). The current Java version is 3.3 (released on October 12, 2014). Encog is released under an Apache 2.0 license.

3.2 How Encog Stores Weights

In their simplest form, Encog stores the weights for a neural network in an array of doubles inside a `FlatNetwork` object.

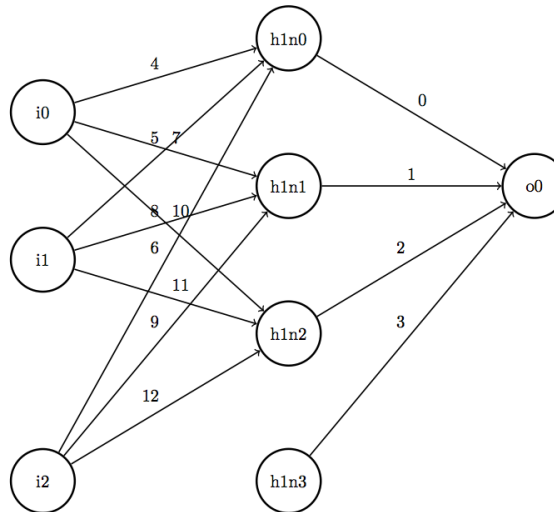


Figure 13: Neural network with labeled weight indexes

As seen in Figure 13, the order of the weights is determined by a combination of the reversed order of the layers and the regular order of the nodes (with the biases being the last node in a layer, if applicable). For example, the network in Figure 13 consists of an input layer of 2 nodes and a bias, a hidden layer of 3 nodes

and a bias, and an output layer of 1 node. The first 4 weights in the array are the weights going from the hidden layer to the output layer (`weights[0]` connects `h1n0` to `o0`, `weights[1]` connects `h1n1` to `o0`...). The next 3 weights connect the input layer to the first hidden node (`weights[4]` connects `i0` to `h1n0`, `weights[5]` connects `i1` to `h1n0`...). This continues in this fashion until the final weight in the array, `weights[12]`, which connects the input bias node to the last regular hidden node (`i2` to `h1n2`).

To access all of the weights at once, the `BasicNetwork` class provides a `dumpWeights()` method. It may also be useful to use the `weightIndex` array from the `FlatNetwork`, which indicates where in the weights array each layer starts.

Alternately, the `BasicNetwork` class has a `getWeight()` method, which allows a user to access the weight from one specific node to another. This is the method that I utilized in my implementation:

```

1  /**
2   * Get the weight between the two layers.
3   * @param fromLayer The from layer.
4   * @param fromNeuron The from neuron.
5   * @param toNeuron The to neuron.
6   * @return The weight value.
7   */
8  public double getWeight(final int fromLayer,
9                          final int fromNeuron,
10                         final int toNeuron) {
11      this.structure.requireFlat();
12      validateNeuron(fromLayer, fromNeuron);
13      validateNeuron(fromLayer + 1, toNeuron);
14      final int fromLayerNumber = getLayerCount() - fromLayer - 1;
15      final int toLayerNumber = fromLayerNumber - 1;
16
17      if (toLayerNumber < 0) {
18          throw new NeuralNetworkError(
19              "The specified layer is not connected to another layer: '"
20              + fromLayer);
21      }
22
23      final int weightBaseIndex
24          = this.structure.getFlat().getWeightIndex()[toLayerNumber];
25      final int count
26          = this.structure.getFlat().getLayerCounts()[fromLayerNumber];
27      final int weightIndex = weightBaseIndex + fromNeuron
28          + (toNeuron * count);
29
30      return this.structure.getFlat().getWeights()[weightIndex];
31  }

```

Figure 14: `BasicNetwork.getWeight()`

3.3 Training

Encog has several different ways to train networks. For the purpose of this project, we will focus on on propagation training.

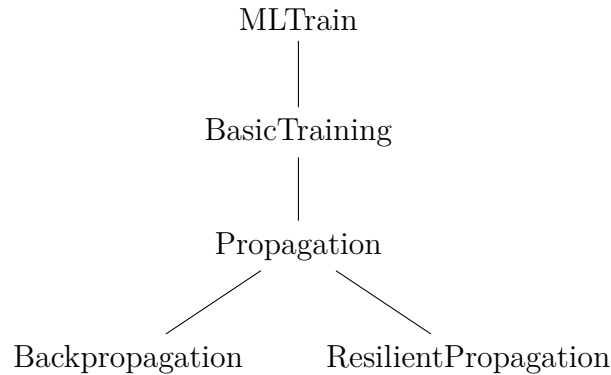


Figure 15: Class hierarchy for training

As seen in figure 15, training in Encog utilizes several different classes. Each method of training that is utilized has its own class (`Backpropagation` and `ResilientPropagation`), with most of the work being done in the parent class `Propagation`. There are other forms of training available, so `Propagation` extends the `BasicTraining` class, and all forms of training must implement the `MLTrain` interface.

Most of the training is done through the `Propagation.iteration()` method, which calls several helper methods. There are two different versions of this method: a default version and a version that accepts the number of iterations as a parameter. In order to do a single iteration, the default form of the method calls the alternate version and passes 1 as a parameter.

The first method to be invoked is `BasicTraining.preIteration()`. This method increments a counter called `iteration`, which keeps track of the current iteration. It also calls upon the `preIteration()` method for any strategies that may be in use.

Strategies are additional methods of training that may be used to enhance

the performance of a training algorithm. The `ResilientPropagation` class doesn't use any, but the `Backpropagation` class allows for the use of two strategies: `SmartLearningRate` and `SmartMomentum`. These strategies will be used to attempt to calculate the learning rate and momentum if they have not been specified upon creation of the network. However, since both of these variables are assigned values by the user (with a default momentum of 0 if the use of that variable is not desired), training strategies are not used in the implementation of this project.

The next method to be invoked is `Propagation.rollIteration()`. However, the use of this method is superfluous. While the `BasicTraining` class has a variable which keeps track of the current iteration, the `Propagation` class has its own copy of that same variable (rather than inheriting the value from its parents class. The `rollIteration()` method increments this duplicate variable. Unfortunately, where the variable in the `BasicTraining` class is utilized in accessor and mutator methods, the same variable in the `Propagation` class is not used anywhere outside of the `rollIteration()` method.

Following this is the `Propagation.processPureBatch()` method (large data sets may want to make use of the `processBatches()` method, which uses a portion of the training set rather than the entire thing). This in turn calls upon `Propagation.calculateGradients()` and `Propagation.learn()`.

`Propagation.calculateGradients()` iterates through the network and calculates the gradient at each portion of the network (for more information, see section 2.1.1). This is done through the `GradientWorker` class. The advantage of this is that it allows for multithreaded calculations. Different portions of the network that don't rely on each other (for example, nodes in the same layer do not have any weights connecting them) can be calculated in parallel using an array of `GradientWorkers`. This project only uses single threaded calculations, so the array has a size of 1.

`Propagation.learn()` uses the gradients to update the weights for the network. Different algorithms update weights in different ways (see sections 2.1.1 and 2.1.2 for more information), so this is an abstract method, with each child class having its own implementation.

The last method to be used is `BasicTraining.postIteration()`. This method calls upon the `postIteration()` method for any strategies if applicable. The `ResilientPropagation` class has its own `postIteration()` method, which stores the error in the `lastError` variable, because RPROP uses this to check for sign changes in the error during subsequent iterations.

3.4 Some Individual Classes

The following sections will go into detail about how I used some of the classes from the Encog library. It wasn't feasible to describe all of the classes used in the program, but these seven were the most important.

3.4.1 TrainingSetUtil

`TrainingSetUtil` (`org.encog.util.simple.TrainingSetUtil`) is the only class that I modified.

The main method that I used was `loadCSVToMemory()`. This method takes a CSV file and loads that into a `ReadCSV` object. Then, that object is converted into something that I could use for training: an `MLDataSet`. There were two problems I was encountering when importing CSV files: incomplete data entries were giving undesired results when training, and an inability to preserve the column headers.

It is not uncommon to have data sets with entries that don't have values for all columns (especially when dealing with data obtained through real world observations). These values can lead to undesired results if used for training, so I wanted to discard those entries in their entirety. Thankfully, attempting to load

empty data values throws a `CSVError` exception (`Error:Unparseable number`), so I was able to surround that part of the code with a try-catch statement. Inside the catch portion, I decided not to print out the stack trace because that information wasn't very useful. However, I did increment a counter I had created called `ignored`, which would then be printed to the console at the conclusion of the importing process.

For the column headers, I needed to create a new data member:

```
private static List<String> columnNames = new ArrayList<String>();
```

The information from the .CSV file is loaded into a `ReadCSV` object. If the .CSV file has headers (as specified through a boolean), these are stored in an `ArrayList` of Strings, which can be accessed through a `getColumnNames()` method in that class. However, there is no way to access that `ReadCSV` object after the initial importing process is completed. Thus, I needed to add some additional functionality to the `TrainingSetUtil` class.

Inside the `loadCSVToMemory()` method, I added a simple statement to store the headers in the data member that I had defined above:

```
if(headers){  
    columnNames = csv.getColumnNames();  
}
```

After that, it was just a matter of creating a standard accessor method (similar to the one in the `ReadCSV` class):

```
/**  
 * @return the columnNames  
 */  
public static List<String> getColumnNames() {
```



```

    return columnNames;
}

```

Back in the main part of the program, I wrote another method to change the ArrayList into a standard array because I am more comfortable accessing information in that format.

Modified Code	Original Code
<pre> 1 if(headers){ 2 columnNames = csv.getColumnNames(); 3 } 4 5 int ignored = 0; 6 7 while (csv.next()) { 8 MLData input = null; 9 MLData ideal = null; 10 int index = 0; 11 try{ 12 input = new BasicMLData(inputSize); 13 for (int i = 0; i < inputSize; i++) { 14 double d = csv.getDouble(index++); 15 input.setData(i, d); 16 } 17 18 if (idealSize > 0) { 19 ideal = new BasicMLData(idealSize); 20 for (int i = 0; i < idealSize; i++) { 21 double d = csv.getDouble(index++); 22 ideal.setData(i, d); 23 } 24 } 25 26 MLDataPair pair = new BasicMLDataPair(input, 27 ideal); 28 result.add(pair); 29 }catch (CSVError e){ 30 ignored++; 31 //e.printStackTrace(); 32 } 33 System.out.println("Rows ignored: " + ignored); 34 return result; </pre>	<pre> 1 2 3 4 5 6 7 while (csv.next()) { 8 MLData input = null; 9 MLData ideal = null; 10 int index = 0; 11 12 input = new BasicMLData(inputSize); 13 for (int i = 0; i < inputSize; i++) { 14 double d = csv.getDouble(index++); 15 input.setData(i, d); 16 } 17 18 if (idealSize > 0) { 19 ideal = new BasicMLData(idealSize); 20 for (int i = 0; i < idealSize; i++) { 21 double d = csv.getDouble(index++); 22 ideal.setData(i, d); 23 } 24 } 25 26 MLDataPair pair = new BasicMLDataPair(input, 27 ideal); 28 result.add(pair); 29 30 31 } 32 33 return result; </pre>

Figure 16: Comparison between modified and original code

Figure 16 shows the differences between the original loadCSVToMemory() method and the modified version. The complete code for this class may be found in Appendix A.6

3.4.2 BasicNetwork

`BasicNetwork` (`org.encog.neural.networks.BasicNetwork`) serves as the main source of interaction between my implementation and the network itself. However, this doesn't necessarily mean that this class does most of the work. Much of the information is stored in related classes (for example, once the format of the network is set up, the majority of information about the network is stored in a `FlatNetwork` object).

Before a network can be used, its structure must be defined. For this purpose, the `BasicNetwork` class uses this data member:

```
/**
 * Holds the structure of the network. This keeps the network from
 * having to
 * constantly lookup layers and synapses.
 */
private final NeuralStructure structure;
```

To set up this structure, each layer must be added through the use of the `addLayer()` method. Each layer passed through the parameters will be added to an `ArrayList` of `Layer` objects. The first layer added will be considered the input layer.

Once all of the layers are added, the network must be finalized by invoking `structure.finalizeStructure()`. Finalizing a neural structure eliminates the intermediate representation of the layers, temporarily storing that information in `FlatLayer` objects, and then creating the `FlatNetwork` object which will be used in the remaining network operations.

Once the network is finalized, the `reset()` method is invoked, which assigns random starting values to the weights.

The actual network training is done through the `Propagation` class (an abstract class which serves as a parent for classes such as `Backpropagation` and `ResilientPropagation`). The `BasicNetwork` object is passed as a parameter, as well as the training data set and any other necessary variables (such as the learning rate and momentum if applicable).

Once the network is fully trained, its effectiveness can be measured by use of the `compute()` method. This is used to compare each ideal output value with the output value the network produces when given the same input.

3.4.3 FlatNetwork

The `FlatNetwork` class (`org.encog.neural.flat.FlatNetwork`) is a more computationally efficient form of a neural network, designed to store everything in single arrays instead of keeping track of everything in multiple layers. Layers are maintained through the use of index arrays, which indicate where each layer starts in the main arrays. According to the javadoc comments, “this is meant to be a very highly efficient feedforward, or simple recurrent, neural network. It uses a minimum of objects and is designed with one principal in mind-- SPEED. Readability, code reuse, object oriented programming are all secondary in consideration”.

In concept, `FlatNetwork` objects act similarly from the standpoint of the user, for they share many of the same methods. However, most of the calculations (such as training) are actually done in this class (the `BasicNetwork` class invokes methods from here). The speed increase comes from the use of single-dimensional arrays of doubles and ints, which have a faster information access time than using accessor and mutator methods with multiple classes.

3.4.4 BasicLayer

The `BasicLayer` class (`org.encog.neural.networks.layers.BasicLayer`) is an implementation of the `Layer` interface. Its job is to store information about the specific layer it is assigned (input, hidden, or output) during the creation of the network. Once the network has been finalized, specific layers are no longer used.

The class has two constructors: one which has user defined parameters (activation function, bias, and neuron count), and one which just receives the number of neurons in the layer. If the second constructor is used, the default option is to create a layer which has a bias and uses a hyperbolic tangent activation function.

Hidden layers utilize all three variables when being initialized. Input layers do not have activation functions. Bias nodes are stored in the layer prior to where they will have an impact (a bias node which effects the nodes in the hidden layer will be declared as part of the input layer), so output layers should not have a bias.

Each layer also has a data member which indicates which network the layer is a part of.

3.4.5 BasicMLDataSet

The `BasicMLDataSet` class (`org.encog.ml.data.basic.BasicMLDataSet`) isn't a very complicated class, but it is very important. A child class for the more general `MLDataSet` interface, the main purpose of this class is to maintain an `ArrayList` of `BasicMLDataPair` objects. This is what the training data set will be stored in. The class contains of several constructors, able to create an object by accepting multidimensional double arrays, an `MLDataSet` object, or an `MLDataPair` `ArrayList`. The rest of the class contains several add methods, as well as methods to retrieve data entries or information about the data set (such as its size).

3.4.6 BasicMLDataPair

The `BasicMLDataPair` class (`org.encog.ml.data.basic.BasicMLDataPair`) is a child class of the `MLDataPair` interface. Its purpose is to hold the information of a single data entry. Each `BasicMLDataPair` contains two `MLData` objects, arrays of doubles designed to store the input data and the ideal data respectively. Both values are necessary for supervised learning, but only the input value is required for unsupervised learning (the ideal value should be left null).

3.4.7 ActivationFunction

`ActivationFunction` (`org.encog.engine.network.activation.ActivationFunction`) is an interface that serves as a parent class for any activation function that would be used with a neural network. The library comes with sixteen activation functions already implemented, but users are free to implement their own as long as they include all of the methods in the interface.

The two most important methods are as follows:

```
void activationFunction(double[] d, int start, int size);
```

This method is the main math portion of the activation function. The input values are stored in the double array `d`, with the range of values specified by the variables `start` and `size`. After some mathematical calculations, the output value from the activation function is stored in the same double array. For example, from the `ActivationSigmoid` class:

```
x[i] = 1.0 / (1.0 + BoundMath.exp(-1 * x[i]));
```

The `ActivationLinear` class actually leaves this method blank. The linear activation function has outputs identical to its inputs, so there is no need to do anything with the array of doubles.

```
double derivativeFunction(double b, double a);
```

This method calculates the derivative of the activation function at a certain point. Not all activation functions have derivatives (there is another method called `hasDerivative()`, which will return true if the specific activation function has a derivative and false otherwise). However, there must be a derivative for an activation function to be used with backpropagation.

The method receives two doubles as parameters. The first double, `b`, is the original input number (in the `activationFunction` method, this number would have been in the `d` array). The second double, `a`, is the original output value. This is the value the activation function produces if it is given `a` as an input. Depending on the equation for each specific activation function, the derivative will be calculated with whichever value is more computationally efficient. For example, the `ActivationSigmoid` class uses the output value:

```
return a * (1.0 - a);
```

To contrast, the `ActivationElliott` class uses the input value:

```
double s = params[0];  
double d = (1.0+Math.abs(b*s));  
return (s*1.0)/(d*d);
```

As of v3.3, all activation functions in the Encog library have derivatives with the exception of `ActivationCompetitive`. Attempting to use this activation function in a backpropagation network will throw an `EncogError` exception (“Can’t use the competitive activation function where a derivative is required”).

CHAPTER 4

Implementation

4.1 Overview

The purpose of this program is to train an artificial neural network and export source code for it. This will allow the results of the network to be used in other projects without needing to store it in a data structure.

All information is controlled through user input via a config file and a training data set. The program will output two files: a training error report, and the code for the network. The exact format of these outputs will be designated by the user.

4.2 Assumptions/Design Choices

Early in the design process, I decided that I was going to use an external third party library to handle the actual training of the neural network. The purpose of this project was more focused on the source code generation for a neural network, rather than the training itself. Doing the actual implementation for the network training would add additional development time to this project. In addition, unless it were to be made the main focus of the project, a personal implementation would not be as effective as a third party alternative, as the designers of said software have spent years optimizing the code. More information about the java library Encog may be found in the previous chapter.

The only other major design decision was the restriction of only numerical information for the training data set. The program is only designed to be used with numbers for all data entries. Using strings will result in rows being ignored when the .csv file is imported. For more information on this decision, see section 5.1.

The program also assumes that all inputs from the user are valid. As of now,

there are very little debugging tools built into the program, so invalid data will result in the program not running.

4.3 Inputs

The program requires two inputs from the user: a config file containing all of the information required by the neural network, and a .csv file containing the training data set.

4.3.1 Config File

The only command line argument is the file path for a config file. This file can have any name, but it must have a .txt file extension. The config file contains the following information:

- The complete file path for the training data set. This file will be described in detail in the next subsection.
- A boolean for whether or not the training data set file has a header row (true for yes, false for no).
- The number of input variables (how many columns in the training data set are independent variables).
- The number of output variables (how many columns in the training data set are dependent variables).
- The number of hidden layers the artificial neural network will be constructed with. There is no theoretical upper limit on the number of hidden layers this program can accommodate, although studies have shown that almost any problem can be solved with the use of at most two hidden layers. [1]
- Attributes for each hidden layer:

- An integer for the type of activation function:
 0. Sigmoid
 1. Hyperbolic Tangent
 2. Linear
 3. Elliott
 4. Gaussian
 5. Logarithmic
 6. Ramp
 7. Sine
 8. Step
 9. Bipolar
 10. Bipolar Sigmoid
 11. Clipped Linear
 12. Elliott Symmetric
 13. Steepened Sigmoid
- A boolean for if the layer has a bias node or not.
- An integer for the number of normal neurons in the layer.
- Attributes for the input layer (only bias information is needed).
- Attributes for the output layer (bias and activation function is needed).
- The file type for the first output file (the training error):
 0. text (.txt)
 1. just numbers (.csv)

- The name of the first output file (not including the file extension; the program will add that information internally).
- The file type for the second output file (the code for the artificial neural network):

0. equation format(.txt)

1. java file (standalone)

2. java file (integrated)

- The name of the second output file (not including the file extension; the program will add that information internally).
- The desired training error. The network will continue to train until the error is less than or equal to this number.
- The maximum number of epochs. If the desired training error has not yet been reached, the network will stop training after this many iterations.
- An integer for the network type:

0. ResilientPropagation (see section 2.1.2)

1. Backpropagation (see section 2.1.1)

- The learning rate. This is only applicable for backpropagation networks.
- The momentum. The program will not use momentum if this value is set to 0. This is only applicable for backpropagation networks.

Comments can be made by beginning a line with a percent symbol (%). The methods related to importing the config file will ignore any such lines.

Rather than prompting the user for this information within the program, using a file allows all of the required information to be stored in one place. This also makes multiple uses of the program easier, because the user is able to change the value of a single variable without going through the tedious process of re-inputting all of the other data as well.

4.3.2 Training Data Set

The other primary input is the training data set. As mentioned in the previous subsection, the file path for this file is given as part of the config file rather than as a command line argument.

The training data set must conform to the following specifications:

- It must be in comma-separated values format (a .csv file).
- Headers are optional. If they are included, the code that the program exports will use the column names as identifiers for variables.
- If possible, do not include any rows with blank entries in them. These rows will be discarded when the .csv file is imported, and therefore not used for training purposes.
- The .csv file shall for organized so that the independent variables (input) are on the left, while the dependent variables (output) are on the right.
- All of the data entries must be numerical. At this time the program does not support categorical classification.

Currently, the program uses the same data set for both training and testing.

4.4 Outputs

The program has two separate output files: one file containing the training error report, and one file containing the code of the neural network.

4.4.1 Training Error Report

The first output file contains information about the training error. This is the overall error (how far the actual results are from the ideal results) after each iteration of training.

The exact format of this file can be specified by the user in the config file. Currently, there are two possible formats.

If the user selects option 0, the output will be in a .txt file:

```
Epoch #1 Error:0.3341111047060686
Epoch #2 Error:0.28750265295383065
Epoch #3 Error:0.26142669721283035
```

Figure 17: Sample from first output file (.txt)

If the user selects option 1, the output will be in a .csv file. This will have a header row, and can be loaded into other programs for analysis (such as graphing):

```
Epoch,Error
1,0.3341111047060686
2,0.28750265295383065
3,0.26142669721283035
```

Figure 18: Sample from first output file (.csv)

4.4.2 Source Code

The second output file contains the source code for the trained neural network. Regardless of what file format this output is in, there will be two main sections to it: variable declaration, and network calculation.

The variable declaration section is a list of all the variables that will be used in the network calculation section, as well as any default values (such as 1.0 for biases). I decided upon the following naming conventions for variables:

- **i** - Input layer (starts at 0)

- **h** - Hidden layer (starts at 1)
- **o** - Output layer (starts at 0)
- **n** - Number of the node within a layer (starts at 0)
- **f** - Indicates which node from the previous layer the link originates (starts at 0)
- **t** - Total (the sum of all f nodes leading to a specified nodes), before being fed to the activation function.
- Lack of an f or a t indicates that this value is the output from an activation function (or a bias node).

If there are headers present in the input file, these will be included as input and output variable names.

The network calculation section is where the specific weight values are utilized in order to write equations that map the specified input values to output values. This allows the function of the trained network to be maintained without needing to store the information in a data structure.

The exact format of this file can be specified by the user in the config file. Currently, there are two possible formats.

If the user selects option 0, the output will be in a .txt file. Variable declarations will just consist of names, and the network calculation section will just be mathematical formulas.

If the user selects option 1 or 2, the output will be in a .java file. Variables will all be declared as doubles (original input and final output variables will be public, and all others will be private). The network calculation section will be inside a method. Everything will also be inside of a class (which shares the same name as the file itself, as specified by the user in the config file).

4.5 Individual classes

The program itself (not counting the modified Encog library) currently consists of five classes. This number may grow in the future if more output source code types were to be implemented.

4.5.1 NeuralGenerator

The `NeuralGenerator` class is the largest class in the program. Most of the work happens here.

The variable declarations are mostly self explanatory, so they will not be discussed here. The comments for each variable can be viewed in Appendix A.1.

After the initial setup, the first thing the program does is import data from the config file, through the `validateConfig()` method. This method goes through the config file line by line (through the use of a helper method, `nextValidLine()`, which ignores any lines that are commented out, as designated by a ‘%’ at the beginning of a line). All information from the config file is stored into data members so it can be accessed by other methods, and is then printed out to the console.

The `initializeOutput1()` method is called, which creates the first output file. This file will contain the training error report. For more information, see section 4.4.1.

The next method to be invoked is `createNetwork()`. This method creates a `BasicNetwork`, and populates it with an input layer, hidden layers, and an output layer. The information for each layer (activation function, bias, and number of nodes) is specified by `LayerInfo` objects, which in turn are defined by the information in the config file. Once all of the layers are added, the network is finalized, and the weights are reset to random values.

Next, the training data set is created from the .csv file. If there are headers, these are stored in an `ArrayList` (the information is then stored in a `String` array,

because I prefer working with that format).

Then, the network is trained. The two current options for training utilize either the `Backpropagation` class or the `ResilientPropagation` class (for more information, see sections 2.1.1 and 2.1.2 respectively). After each iteration of training, the training error is calculated, and written to a file through the `writeOne()` method. This helper method also prints the information to the console. Training will continue until the training error is below the desired amount, or until the maximum number of epochs has been reached.

Once the network is trained, the first file is closed. The program prints the results of the network to the console, comparing the actual value of each input to its ideal value.

Finally, the `initializeOutput2()` method is invoked. This method creates the code output file (see section 4.4.2), and stores the necessary values in variables through accessor methods in the `OutputWriter` class. Finally, the program flow then proceeds to the `writeFile()` method for the desired `OutputWriter` child class, and then the program terminates.

4.5.2 LayerInfo

`LayerInfo` is a small class created to store the information needed to create a layer in an artificial neural network. I had originally planned on using a struct, but java does not support those, so I decided to make a separate class to hold the same information.

The class has 3 main variables:

- **private int** `activationFunction` - An integer for the type of activation function for that layer.
- **private boolean** `isBiased` - A boolean for whether or not the layer has a bias

node.

- **private int** neurons - An integer for the number of normal (non-bias) neurons in the layer.

All of these variables are set through parameters passed to the constructor. There should not be a need to change these values once they have been set, so there are no mutator methods. Each variable has an accessor method so that its value can be used by the rest of the program.

The only other method is the `toString()` method. This method is used for returning the information from the layer in an easy-to-read format, so that it can be printed. While not essential to the flow of the program, it may be useful for the user to see this information displayed in the console (especially for debugging purposes).

4.5.3 OutputWriter

The `OutputWriter` class serves as a parent class for other `OutputWriters`. This class holds all of the shared methods required to create a file and output the code/formula for a trained artificial neural network.

Child classes must implement three methods: `createFile()`, `writeFile()`, and `parseActivationFunction()`.

The `createFile()` method creates the file used for output. While the majority of the code in this method is the same in all child classes, I found that it was easier to have each child class add its own file extension to the file name (.txt or .java).

The `writeFile()` method is rather lengthy. This is where the actual program writes the code/formula for the neural network to a file. While similar in terms of basic structure, the actual details of this will vary with each child class.

The `parseActivationFunction()` parses the equation of the activation function

and returns it in String form. A series of **if-else** statements allowed for 14 of the 16 currently implemented activation functions to be used (Softmax is rather complicated and would require the code to be reworked, and Competitive is non-differentiable so I didn't see a need to include it).

4.5.4 **OutputWriterTxt**

The `OutputWriterTxt` class is a child of the `OutputWriter` class. This class will be used if the user selects option 0 for the second output file.

The `createFile()` method creates the second output file, using the filename as specified in the config file and appending a `.txt` file extension.

The variable declarations section gives the names of all the variables to be used in the network calculation section, in the following order:

- Header names for input (if applicable).
- Input layer.
- Hidden layer(s).
- Output layer.
- Header names for output (if applicable).

If there any are bias nodes present, they are assigned the value of the bias as defined in the network (the default is 1.0, but this value is customizable).

Following the variable declaration is the network calculation section. If there are any variables defined by header names, the values of these are stored into the predefined variables such as `i0`. After that, calculation begins starting with the first hidden layer. The `f` values are calculated first, consisting of the associated node in the previous layer multiplied by the weight of the connection (as defined by the trained network). All `f` values for a specific node are added together, with

the resulting sum being stored in a `t` value. This `t` value is then fed through the activation function (the text of which comes from the `parseActivationFunction()` method), and that is stored in the final value for that node. This continues through all of the hidden layers and the output layer. Finally, if applicable, the values of the final outputs are stored in the variables defined by output header names.

The `parseActivationFunction()` method parses the equation of the activation function and returns it in String form. All information with regards to the exact mathematical formulas for each activation function came from the `activationFunction()` method of the associated class.

```
//Variable declarations
//Header Names
x1
x2
xor
//Input layer
i0          i0 = x1
i1          i1 = x2
i2 = 1.0
//Hidden layer(s)
//Hidden Layer 1
h1n0f0      h1n0f0 = i0 * 1.367776537359666
h1n0f1      h1n0f1 = i1 * 1.2920909769370108
h1n0f2      h1n0f2 = i2 * -2.3097219906621054
h1n0t       h1n0t = h1n0f0 + h1n0f1 + h1n0f2
h1n0        h1n0 = 1.0 / (1.0 + e^(-1 * h1n0t))
h1n0f0      h1n1f0 = i0 * 4.975164727376538
h1n0f1      h1n1f1 = i1 * 4.634590911948785
h1n0f2      h1n1f2 = i2 * -1.1798243603241103
h1n1t       h1n1t = h1n1f0 + h1n1f1 + h1n1f2
h1n1        h1n1 = 1.0 / (1.0 + e^(-1 * h1n1t))
h1n2 = 1.0
//Output layer
o0f0        o0f0 = h1n0 * -15.158855977843674
o0f1        o0f1 = h1n1 * 10.890253618352084
o0f2        o0f2 = h1n2 * -3.360679967779563
o0t         o0t = o0f0 + o0f1 + o0f2
o0          o0 = 1.0 / (1.0 + e^(-1 * o0t))
xor         xor = o0
```

(a)

(b)

Figure 19: Sample second output file (.txt)

4.5.5 OutputWriterJava

The `OutputWriterJava` class is a child of the `OutputWriter` class. This class will be used if the user selects option 1 or 2 for the second output file.

The `createFile()` method creates the second output file, using the filename as specified in the config file and appending a .java file extension.

The format of the .java file was inspired by the output of the program Tiberius. One of the original intents of this program was to be used in a course that currently uses Tiberius, so it made sense to model the output file in a way that it would be compatible.

The first things written to the file is an import statement for `java.lang.Math`, followed by the declaration of the class (with the same name as the file).

The variable declarations section declares all of the variables to be used in the network calculation section, in the same order as specified in the previous subsection. All methods are static so that they can be accessed from the main method without creating a specific object of this class type, so all variables are declared as static doubles. Most variables are private, but the input and output variables (as well as any variable names defined by headers in the training data set) are declared as public so that they can be accessed by the user in other classes. If there any are bias nodes present, they are assigned the value of the bias as defined in the network (the default is 1.0, but this value is customizable). Bias nodes are also always declared as private, even if they are in the input layer.

If the user has chosen to make a standalone java file, there will be two additional methods: `main()` and `initData()`. The `main()` method will call the other two methods (`initData()` and `calcNet()`), and then print the output values to the console. The `initData()` method will provide default values for the input variables (using the header names if applicable). The default values are currently set to 1, although these can be modified by the user. If the user has selected to make an integrated java file, neither of these two methods will be present.

The `calcNet()` method contains the network calculation section of the code. The code generation of this section is almost identical to the equivalent in the `OutputWriterTxt` class, except that every line ends with a semicolon.

The `parseActivationFunction()` method parses the equation of the activation function and returns it in String form. Some mathematical expressions are substituted with their java equivalents (for example, $|x|$ becomes `Math.abs(x)`, and e^x becomes `Math.exp(x)`).

```
import java.lang.Math;

public class output2
{
    //Variable declarations
    //Header Names
    public static double x1;
    public static double x2;
    public static double xor;
    //Input layer
    public static double i0;
    public static double i1;
    private static double i2 = 1.0;
    //Hidden layer(s)
    //Hidden Layer 1
    private static double h1n0f0;
    private static double h1n0f1;
    private static double h1n0f2;
    private static double h1n0t;
    private static double h1n0;
    private static double h1n1f0;
    private static double h1n1f1;
    private static double h1n1f2;
    private static double h1n1t;
    private static double h1n1;
    private static double h1n2 = 1.0;
    //Output layer
    private static double o0f0;
    private static double o0f1;
    private static double o0f2;
    private static double o0t;
    public static double o0;

    public static void main(String[] args)
    {
        initData();
        calcNet();
        System.out.println(o0);
    }
}

public static void initData()
{
    //data is set here
    x1 = 1;
    x2 = 1;
}

public static void calcNet()
{
    i0 = x1;
    i1 = x2;

    h1n0f0 = i0 * -918.089072207438;
    h1n0f1 = i1 * 3.998008385237878;
    h1n0f2 = i2 * -8.631004060244248;
    h1n0t = h1n0f0 + h1n0f1 + h1n0f2;
    h1n0 = 1.0 / (1.0 + Math.exp(-1 * h1n0t));

    h1n1f0 = i0 * 3.0491035435976457;
    h1n1f1 = i1 * -297.45021956037084;
    h1n1f2 = i2 * -6.180183917718583;
    h1n1t = h1n1f0 + h1n1f1 + h1n1f2;
    h1n1 = 1.0 / (1.0 + Math.exp(-1 * h1n1t));

    o0f0 = h1n0 * 507.0365297905731;
    o0f1 = h1n1 * 132.9962652623966;
    o0f2 = h1n2 * -2.1795899783064443;
    o0t = o0f0 + o0f1 + o0f2;
    o0 = 1.0 / (1.0 + Math.exp(-1 * o0t));

    xor = o0;
}
}
```

(a)
(b)

Figure 20: Sample second output file (.java)

List of References

- [1] J. Heaton. "The number of hidden layers." September 2008. [Online]. Available: <http://www.heatonresearch.com/node/707>

CHAPTER 5

Future work and conclusions

5.1 Categorical classification

As of right now, the program only works with data sets with entirely numerical entries. This means that any data sets with categorical entries will need to be changed into a numerical representation before they can be used. For example, with the iris data set, instead of species of setosa, versicolor, and virginica, it would use numbers such as 1, 2, and 3.

My original concept was to start out with numerical classification first, because it is easier to work with, and then expand to include categorical if I had time. However, I discovered late in the implementation period that in order to be able to use categorical data sets, I would have to completely change how the network itself was implemented. Within Encog, categorical classification is done with several different classes than numerical classification.

As of the writing of this paper, I do not know if I can use those classes to work with numerical data sets, or if I would have to make a separate main class for the different types.

5.2 Additional output formats

Originally, this project was going to be written in C++, because of the applicability of that language to the gaming industry (where I want to work)[1]. However, it was changed to Java because of the portability of that language (there is no need to compile the code for different systems, because it is always run within the java virtual machine).

Currently, the second output file from the program can be in either basic equational format (in a .txt file), or in Java code. Given more time, I would

have preferred to also allow for C++ code to be exported. Given the nature of the program, it would not be unfeasible to allow other target languages to be implemented as well.

5.3 Non-command line inputs

Currently, the only input into the program is through a config file, which contains all of the necessary data that the program needs to run. While this can be easier for multiple runs (because the user does not need to repeatedly input the information each time), I recognize that it can be hard to set up the config file for the first time. Some users may also prefer to enter the information on a step by step basis.

The basic implementation of an alternate input method is not very complicated. If there are no arguments passed to the program when it is run, a new method would be called instead of the `validateConfig()` method. This method would assign values for all of the necessary variables through a series of input statements utilizing a scanner.

Related to this additional input method would be improved config file debugging. Currently the program assumes that all of the data the user has inputted is valid. There is no checking in that method to see if a number is within a valid range (for example, a value of 17 for the activation function type). These numbers are checked in other places of the code, but it would be more useful for the user to have the numbers validated the first time they are encountered. If there is a major problem (for example, the program is expecting a number, and the user has a string of text instead), a generic exception will be thrown, and the stack trace will be printed to the console.

Although these implementations are not very difficult, they were omitted for because of timing.

5.4 Normalization

Depending on the data set being used, normalization can be an important feature in machine learning. For example, data sets with values for a certain variable that are much larger than other values may not converge as well during training as with a normalized data set. Encog supports several different types of normalization, but I would most likely be using range normalization due to the ability to normalize the data to a specific range (for example, -1 to 1 when using a hyperbolic tangent activation function, or 0 to 1 when using a sigmoid activation function).

$$f(x) = \frac{(x-d_L)(n_H-n_L)}{(d_H-d_L)} + n_L \quad [2]$$

In this equation, d_L is the minimum value (low) in the data set, d_H is the maximum value (high) in the data set, n_L is the desired normalized minimum, and n_H is the desired normalized maximum.

5.5 Conclusions

Overall, the program works. The best way to illustrate this is to walk through an example.

In this example, the config file shown in figure 27 was used. This creates a network with a single hidden row of 2 neurons, a sigmoid function for an activation function, and RPROP for a training algorithm. The XOR function was used as a simple data set, as seen in figure 21.

x1	x2	xor
0	0	0
1	0	1
0	1	1
1	1	0

Figure 21: test.csv

Due to the speed of RPROP, this network was able to train to an error of 0.01 within 47 iterations. A .csv format was chosen for the first output file, which can

be seen in figure 22.

Epoch	Error	16	0.24331544	32	0.10324324
1	0.27828625	17	0.24145497	33	0.10371875
2	0.26261123	18	0.23910242	34	0.09008032
3	0.25154451	19	0.23688016	35	0.08526512
4	0.25082551	20	0.23280682	36	0.07503029
5	0.2514136	21	0.2280602	37	0.06615781
6	0.24935344	22	0.22064924	38	0.06183121
7	0.25124387	23	0.21124914	39	0.05247069
8	0.24911844	24	0.19860283	40	0.04412141
9	0.24879488	25	0.18573386	41	0.03573549
10	0.24863693	26	0.16598462	42	0.0310084
11	0.24818165	27	0.14295186	43	0.0238079
12	0.24760513	28	0.1266399	44	0.01915438
13	0.24695005	29	0.15851361	45	0.01462404
14	0.24600628	30	0.12725001	46	0.01027347
15	0.24481026	31	0.11714096	47	0.00673701

(a)
(b)
(c)

Figure 22: output1.csv

CSV files can easily be plotted, as seen in figure 23. The spike at epoch 29 most likely illustrates the nature of the algorithm to correct itself after skipping past a local minimum, as denoted by a sign change in the gradient.

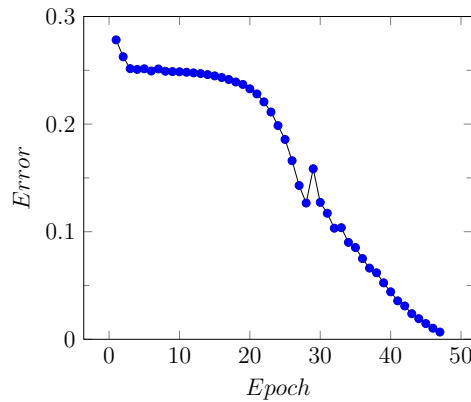


Figure 23: graph of output1.csv

The second output is in a java file, as seen in figure 28. The integrated option was chosen, so there will not be any main method.

The program also printed the results of testing the network, as seen in figure 24.


```

Neural Network Results:
0.0,0.0, actual=0.052266318254488506,ideal=0.0
1.0,0.0, actual=0.9465544658147504,ideal=1.0
0.0,1.0, actual=0.9092496344940679,ideal=1.0
1.0,1.0, actual=0.05052107883471591,ideal=0.0

```

Figure 24: Results from NeuralGenerator.java

In order to test to ensure that the program outputted source code correctly, I wrote the following short program, seen in figure 25. This program tests the code for the network by using all four values of the original training data set, and outputting the results.

```

1 package test;
2
3 public class TestModule {
4
5     private static int[][] tests =
6         {{0,0},{1,0},{0,1},{1,1}};
7
8     public static void main(String[] args) {
9
10        for (int i = 0; i < 4; i++){
11            System.out.print("\nTest #" + i + ": (" +
12                tests[i][0]
13                + "," + tests[i][1] + ") \t\t Results: ");
14            output2.x1 = tests[i][0];
15            output2.x2 = tests[i][1];
16            output2.calcNet();
17            System.out.print(output2.xor);
18        }
19    }
20 }

```

Figure 25: TestModule.java

When this program is run, it produces the output seen in figure 26.

```

Test #0: (0,0)      Results: 0.052266318254488506
Test #1: (1,0)      Results: 0.9465544658147504
Test #2: (0,1)      Results: 0.9092496344940679
Test #3: (1,1)      Results: 0.05052107883471591

```

Figure 26: Results from output2.java

By comparing the results from figures 24 and 26, it is evident that the generated source code produces the same results as the trained network. Therefore, the network has been successfully maintained without the use of additional data structures.

There are always improvements that can be made (for example, those listed in sections 5.1-5.4). However, considering the initial scope of the project, the program that has been created can be considered successful.

<pre>% Example Config file % complete file address for dataset, in .csv format /Users/bwinrich/Documents/workspace/Thesis/test.csv % Does the cvs file have headers? true % Number of input parameters 2 % Number of output parameters 1 % Number of hidden layers 1 % Attributes of layers % Use the following format: (activationFuction, isBiased, neurons), % Each layer should be on its own line % activationFuction is an int, with the following activation functions available: %% 0 - Sigmoid %% 1 - Hyperbolic Tangent %% 2 - Linear %% 3 - Elliott %% 4 - Gaussian %% 5 - Logarithmic %% 6 - Ramp %% 7 - Sine %% 8 - Step %% 9 - BiPolar %% 10 - BiPolar Sigmoid %% 11 - Clipped Linear %% 12 - Elliott Symmetric %% 13 - Steepened Sigmoid % isBiased is a boolean (true if the layer has a bias, false otherwise) % neurons is an int (the number of neurons in that layer) (0, true, 2) % Finally, the attributes of the first and last layer % The input layer doesn't have an activation function, % so we only need to know if it is biased (true)</pre>	<pre>% The output layer has an activation function, % but we don't need to know how many neurons it has, % Or if it has a bias (0) % Output 1 file type %% 0 = text (txt) %% 1 = just numbers (csv) 1 % Output 1 file name (do not include file extension at the end) output1 % Output 2 file type %% 0 = equation format (.txt file) %% 1 = java file (standalone) %% 2 = java file (integrated) 2 % Output 2 file name (do not include file extension at the end) output2 % Desired training error (the network will train until % the error is less than or equal to this) 0.01 % Maximum number of epochs 10000 % Network Type %% 0 = ResilientPropagation %% 1 = Backpropagation 0 % Learning Rate % (only applicable for Backpropagation) 0.7 % Momentum (if you do not want to use momentum, set this to 0) % (only applicable for Backpropagation) 0</pre>
(a)	(b)

Figure 27: Sample config file

List of References

- [1] K. Stuart. "How to get into the games industry – an insiders' guide." March 2014. [Online]. Available: <http://www.theguardian.com/technology/2014/mar/20/how-to-get-into-the-games-industry-an-insiders-guide>
- [2] J. Heaton. "Range normalization." September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Range_Normalization

```

import java.lang.Math;

public class output2
{
    //Variable declarations
    //Header Names
    public static double x1;
    public static double x2;
    public static double xor;
    //Input layer
    public static double i0;
    public static double i1;
    private static double i2 = 1.0;
    //Hidden layer(s)
    //Hidden Layer 1
    private static double h1n0f0;
    private static double h1n0f1;
    private static double h1n0f2;
    private static double h1n0t;
    private static double h1n0;
    private static double h1n1f0;
    private static double h1n1f1;
    private static double h1n1f2;
    private static double h1n1t;
    private static double h1n1;
    private static double h1n2 = 1.0;
    //Output layer
    private static double o0f0;
    private static double o0f1;
    private static double o0f2;
    private static double o0t;
    public static double o0;

    public static void calcNet()
    {
        i0 = x1;
        i1 = x2;

        h1n0f0 = i0 * 3.6869200533700797;
        h1n0f1 = i1 * -3.3847620146833983;
        h1n0f2 = i2 * -2.0233813712749544;
        h1n0t = h1n0f0 + h1n0f1 + h1n0f2;
        h1n0 = 1.0 / (1.0 + Math.exp(-1 * h1n0t));

        h1n1f0 = i0 * 8.161271474645059;
        h1n1f1 = i1 * -7.657582392497621;
        h1n1f2 = i2 * 1.8305055444152039;
        h1n1t = h1n1f0 + h1n1f1 + h1n1f2;
        h1n1 = 1.0 / (1.0 + Math.exp(-1 * h1n1t));

        o0f0 = h1n0 * 9.36214064231166;
        o0f1 = h1n1 * -7.281180690756561;
        o0f2 = h1n2 * 2.284144577336713;
        o0t = o0f0 + o0f1 + o0f2;
        o0 = 1.0 / (1.0 + Math.exp(-1 * o0t));

        xor = o0;
    }
}

```

(a)
(b)

Figure 28: output2.java

APPENDIX

Source Code

A.1 NeuralGenerator.java

```
1  package skynet;
2  import org.encog.Encog;
3  import org.encog.engine.network.activation.*;
4  import org.encog.ml.data.MLData;
5  import org.encog.ml.data.MLDataPair;
6  import org.encog.ml.data.MLDataSet;
7  import org.encog.neural.networks.BasicNetwork;
8  import org.encog.neural.networks.layers.BasicLayer;
9  import org.encog.neural.networks.training.propagation.Propagation;
10 import org.encog.neural.networks.training.propagation.back.Backpropagation;
11 import org.encog.neural.networks.training.propagation.resilient.*;
12 import org.encog.util.csv.CSVFormat;
13 import org.encog.util.simple.TrainingSetUtil;
14 import java.io.BufferedReader;
15 import java.io.BufferedWriter;
16 import java.io.File;
17 import java.io.FileInputStream;
18 import java.io.FileWriter;
19 import java.io.IOException;
20 import java.io.InputStreamReader;
21 import java.util.List;
22
23 /**
24  * The main class for the program. The purpose of this program is the train
25  * an Artificial Neural Network and output source code for it, so that it
26  * can be used in other projects.
27  * @author bwinrich
28  *
29  */
30 public class NeuralGenerator {
31
32     /**
33      * A BufferedWriter for our error output
34      */
35     private BufferedWriter bw1 = null;
36
37     /**
38      * An OutputWriter for our code output
39      */
40     private OutputWriter myOutput;
```

```

41
42  /**
43   * The number of neurons in each layer (including bias neurons)
44   */
45  private int[] numberOfTotalNeurons;
46
47  /**
48   * The number of neurons in each layer (excluding bias neurons)
49   */
50  private int[] numberOfNormalNeurons;
51
52  /**
53   * The location of the .csv file for the training data set
54   */
55  private String filePath = null;
56
57  /**
58   * Does the data set have headers?
59   */
60  private boolean hasHeaders = false;
61
62  /**
63   * The number of input nodes
64   */
65  private int numOfInput = 0;
66
67  /**
68   * The number of output nodes
69   */
70  private int numOfOutput = 0;
71
72  /**
73   * The number of hidden layers in the network
74   */
75  private int numOfHiddenLayers = 0;
76
77  /**
78   * An array holding the information for each layer (activation function,
79   * bias, number of neurons)
80   */
81  private LayerInfo[] allMyLayers = null;
82
83  /**
84   * The network will train until the error is this value or lower
85   */
86  private double desiredTrainingError = 0.01;

```

```

87
88  /**
89   * The maximum number of epochs the network will train
90   */
91  private int numOfEpochs = 0;
92
93  /**
94   * The learning rate for the network (backpropagation only)
95   */
96  private double learningRate = 0;
97
98  /**
99   * The momentum for the network (backpropagation only)
100   */
101  private double momentum = 0;
102
103  /**
104   * The type of file the error output will be (0: .txt, 1: .csv)
105   */
106  private int output1Type = 0;
107
108  /**
109   * The type of file the code output will be (0: .txt, 1/2: .java)
110   */
111  private int output2Type = 0;
112
113  /**
114   * The name of the error output file
115   */
116  private String output1Name = null;
117
118  /**
119   * The name of the code output file
120   */
121  private String output2Name = null;
122
123  /**
124   * The data structure for the ANN
125   */
126  private BasicNetwork network = null;
127
128  /**
129   * Array to hold the column names (only if the data set has headers)
130   */
131  private String[] columnNames;
132

```

```

133  /**
134   * The type of network the ANN will be (0: Resilient propagation,
135   * 1: backpropagation)
136   */
137  private int networkType;
138
139  /**
140   * The training data set
141   */
142  private MLDataSet trainingSet;
143
144
145  /**
146   * The main method.
147   * @param args Should contain the location of the config file
148   */
149  @SuppressWarnings("unused")
150  public static void main(final String args[]) {
151      if (args.length == 0){
152          System.out.println("Error: No file");
153      }else{
154          String configFilePath = args[0];
155          NeuralGenerator myThesis = new NeuralGenerator(configFilePath);
156      }
157  }
158
159  /**
160   * Constructor for the class.
161   * @param configFilePath The location of the config file
162   */
163  public NeuralGenerator(String configFilePath){
164      newMain(configFilePath);
165  }
166
167  /**
168   * The driver method, which will call all other necessary methods
169   * required for the execution of the program
170   * @param configFilePath The location of the config file
171   */
172  private void newMain(String configFilePath){
173
174      //Import the config file, and the necessary information
175      validateConfig(configFilePath);
176
177      //Create the first output file
178      System.out.println("Initializing first output file...");

```

```

179 initializeOutput1();
180
181 // create a neural network
182 System.out.println("Creating network...");
183 createNetwork();
184
185 //Import data set
186 System.out.println("Importing csv file...");
187
188 trainingSet = TrainingSetUtil.loadCSVToMemory(CSVFormat.ENGLISH,
189     filePath, hasHeaders, numOfInput, numOfOutput);
190
191 //Just because I prefer working with arrays instead of arrayList
192 if(hasHeaders){
193     List<String> columns = TrainingSetUtil.getColumnNames();
194     int width = columns.size();
195     columnNames = new String[width];
196     for (int i = 0; i < width; i++){
197         columnNames[i] = columns.get(i);
198     }
199 }
200
201 // train the neural network
202 train();
203
204 // Close the first file after we're done with it
205 try{
206     if(bw1!=null)
207         bw1.close();
208 }catch(Exception ex){
209     System.out.println("Error in closing the BufferedWriter"+ex);
210 }
211
212 // test the neural network
213 System.out.println("");
214 System.out.println("Neural Network Results:");
215 for(MLDataPair pair: trainingSet ) {
216     final MLData output = network.compute(pair.getInput());
217
218     String printInput = "";
219     String printOutput = "";
220     String printIdeal = "";
221
222     //Iterate through the input data to generate a string.
223     for(int i = 0; i < numOfInput; i++){
224         printInput += pair.getInput().getData(i);

```



```

225         printInput += ",";
226     }
227
228     //We do the same thing with the output data
229     for(int i = 0; i < numOfOutput; i++){
230         printOutput += output.getData(i);
231         printOutput += ",";
232     }
233
234     //And the same thing for the ideal data
235     for(int i = 0; i < numOfOutput; i++){
236         printIdeal += pair.getIdeal().getData(i);
237         if((i+1) != numOfOutput){
238             printIdeal += ",";
239         }
240     }
241
242     System.out.println(printInput + " actual=" + printOutput
243         + " Ideal=" + printIdeal);
244 }
245
246 //Some additional numbers that we need
247 int layers = network.getLayerCount();
248
249 numberOfTotalNeurons = new int [layers];
250 numberOfNormalNeurons = new int [layers];
251
252 for (int i = 0; i<layers; i++)
253 {
254     numberOfTotalNeurons[i] = network.getLayerTotalNeuronCount(i);
255     numberOfNormalNeurons[i] = network.getLayerNeuronCount(i);
256 }
257
258 System.out.println("\n");
259
260 //Initialize the OutputWriter
261 System.out.println("Initializing Second Output File...");
262 initializeOutput2();
263
264 System.out.println("Writing to file...");
265
266 myOutput.writeFile();
267
268 System.out.println("Done.");
269
270 Encog.getInstance().shutdown();

```

```

271     }
272
273
274     /**
275     * This method handles the training of the Artificial Neural Network
276     */
277     private void train() {
278         Propagation train = null;
279
280         //Different networks will be created based on the type listed in the
281         //config file
282         switch(networkType){
283             case 0:
284                 train = new ResilientPropagation(network, trainingSet);
285                 break;
286             case 1:
287                 train = new Backpropagation(network, trainingSet, learningRate,
288                     momentum);
289                 break;
290             default:
291                 break;
292         }
293
294         int epoch = 1;
295
296         System.out.println("");
297
298         System.out.println("Training...");
299
300         System.out.println("");
301
302         //Training the network
303         do {
304             train.iteration();
305
306             //We write the error to the first output file
307             writeOne(epoch, train.getError());
308
309             epoch++;
310         } while((train.getError() > desiredTrainingError)
311             && (epoch < numOfEpochs));
312         //Training will continue until the error is not above the desired
313         //error, or until the maximum number of epochs has been reached
314         train.finishTraining();
315     }
316

```

```

317  /**
318   * Helped method for creating the ANN and adding layers to it
319   */
320  private void createNetwork() {
321      network = new BasicNetwork();
322
323      for (LayerInfo myLayer:allMyLayers)
324      {
325          switch (myLayer.getActivationFunction()){
326              case -1: //The input layer doesn't have an activation function
327                  network.addLayer(new BasicLayer(null, myLayer.isBiased(),
328                      myLayer.getNeurons()));
329                  break;
330              case 0:
331                  network.addLayer(new BasicLayer(new ActivationSigmoid(),
332                      myLayer.isBiased(), myLayer.getNeurons()));
333                  break;
334              case 1:
335                  network.addLayer(new BasicLayer(new ActivationTANH(),
336                      myLayer.isBiased(), myLayer.getNeurons()));
337                  break;
338              case 2:
339                  network.addLayer(new BasicLayer(new ActivationLinear(),
340                      myLayer.isBiased(), myLayer.getNeurons()));
341                  break;
342              case 3:
343                  network.addLayer(new BasicLayer(new ActivationElliott(),
344                      myLayer.isBiased(), myLayer.getNeurons()));
345                  break;
346              case 4:
347                  network.addLayer(new BasicLayer(new ActivationGaussian(),
348                      myLayer.isBiased(), myLayer.getNeurons()));
349                  break;
350              case 5:
351                  network.addLayer(new BasicLayer(new ActivationLOG(),
352                      myLayer.isBiased(), myLayer.getNeurons()));
353                  break;
354              case 6:
355                  network.addLayer(new BasicLayer(new ActivationRamp(),
356                      myLayer.isBiased(), myLayer.getNeurons()));
357                  break;
358              case 7:
359                  network.addLayer(new BasicLayer(new ActivationSIN(),
360                      myLayer.isBiased(), myLayer.getNeurons()));
361                  break;
362              case 8:

```

```

363         network.addLayer(new BasicLayer(new ActivationStep(),
364             myLayer.isBiased(), myLayer.getNeurons()));
365         break;
366     case 9:
367         network.addLayer(new BasicLayer(new ActivationBiPolar(),
368             myLayer.isBiased(), myLayer.getNeurons()));
369         break;
370     case 10:
371         network.addLayer(new BasicLayer(
372             new ActivationBipolarSteepenedSigmoid(),
373             myLayer.isBiased(), myLayer.getNeurons()));
374         break;
375     case 11:
376         network.addLayer(new BasicLayer(new ActivationClippedLinear(),
377             myLayer.isBiased(), myLayer.getNeurons()));
378         break;
379     case 12:
380         network.addLayer(new BasicLayer(new ActivationElliottSymmetric(),
381             myLayer.isBiased(), myLayer.getNeurons()));
382         break;
383     case 13:
384         network.addLayer(new BasicLayer(new ActivationSteepenedSigmoid(),
385             myLayer.isBiased(), myLayer.getNeurons()));
386         break;
387     default:
388         //Unimplemented activation function: Softmax (complicated)
389         //Unimplemented activation function: Competitive
390         //(non-differentiable)
391         System.out.println("Error: This activation function is "
392             + "either invalid or not yet implemented");
393         break;
394     }
395 }
396
397 network.getStructure().finalizeStructure();
398
399 network.reset();
400 }
401
402
403 /**
404  * This method creates the error output file
405  */
406 private void initializeOutput1() {
407     String output1NameFull = null;
408

```

```

409     //File type is specified in the config file
410     switch(output1Type){
411     case 0:
412         output1NameFull = output1Name + ".txt";
413         break;
414     case 1:
415         output1NameFull = output1Name + ".csv";
416         break;
417     default:
418         //More cases can be added at a later point in time
419         System.out.println("Invalid output 1 type");
420     }
421
422     try{
423         File file1 = new File(output1NameFull);
424
425         if (!file1.exists()) {
426             file1.createNewFile();
427         }
428
429         FileWriter fw1 = new FileWriter(file1);
430         bw1 = new BufferedWriter(fw1);
431
432         //Header line for a .csv file
433         if (output1Type == 1){
434             bw1.write("Epoch,Error");
435             bw1.newLine();
436         }
437     }catch (IOException e){
438         // TODO Auto-generated catch block
439         e.printStackTrace();
440     }
441 }
442
443 /**
444  * This method creates the code output file
445  */
446 private void initializeOutput2(){
447
448     //File type is specified in the config file
449     switch(output2Type){
450     case 0:
451         myOutput = new OutputWriterTxt();
452         break;
453     case 1:
454         myOutput = new OutputWriterJava(true);

```

```

455         break;
456     case 2:
457         myOutput = new OutputWriterJava(false);
458         break;
459     default:
460         //More cases can be added if additional classes are designed
461         System.out.println("Invalid output 2 type");
462         break;
463     }
464
465     //Creating the file
466     myOutput.createFile(output2Name);
467
468     //Passing all of the necessary network information to the OutputWriter
469     myOutput.setNetwork(network);
470     myOutput.setInputCount(numOfInput);
471     myOutput.setOutputCount(numOfOutput);
472     myOutput.setLayers(numOfHiddenLayers+2);
473     myOutput.setNumberOfTotalNeurons(numberOfTotalNeurons);
474     myOutput.setNumberOfNormalNeurons(numberOfNormalNeurons);
475     myOutput.setHasHeaders(hasHeaders);
476     myOutput.setColumnNames(columnNames);
477     myOutput.initializeOtherVariables();
478 }
479
480 /**
481  * Helper method for writing to the error output file
482  * @param epoch Number of times the network has been trained
483  * @param error Training error for that epoch
484  */
485 private void writeOne(int epoch, double error) {
486
487     String temp = null;
488
489     //Format depends on file type
490     switch(output1Type){
491     case 0:
492         temp = "Epoch #" + epoch + " Error:" + error;
493         break;
494     case 1:
495         temp = "" + epoch + "," + error;
496         break;
497     default:
498         temp = "Invalid output 2 type";
499         break;
500     }

```

```

501
502     //Output the error to the console before writing it to the file
503     System.out.println(temp);
504     try {
505         bw1.write(temp);
506         bw1.newLine();
507     } catch (IOException e) {
508         // TODO Auto-generated catch block
509         e.printStackTrace();
510     }
511 }
512
513 /**
514  * Helper method for retrieving lines from the config file. Comments are
515  * not considered valid lines.
516  * @param d The BufferedReader for the config file
517  * @return The next valid line from the config file
518  * @throws IOException
519  */
520 private String nextValidLine(BufferedReader d) throws IOException
521 {
522     String validLine = null;
523     boolean isValid = false;
524
525     if (d.ready()){
526         do{
527             String str = d.readLine();
528
529             if (str.length() != 0){
530                 //Eliminate extra space
531                 str = str.trim();
532
533                 //Comments start with %, and are not considered valid
534                 if (str.charAt(0) != '%'){
535
536                     validLine = str;
537                     isValid = true;
538                 }
539             }
540         }while (!isValid && d.ready());
541     }
542     return validLine;
543 }
544
545 /**
546  * A lengthy method for validating the config file. All information from

```

```

547 * the config file is stored into data members so it can be accessed by
548 * other methods.
549 * @param configFilepath The location of the config file
550 */
551 public void validateConfig(String configFilepath)
552 {
553     try{
554         File myFile = new File(configFilepath);
555         FileInputStream fis = null;
556
557         BufferedReader d = null;
558
559         fis = new FileInputStream(myFile);
560
561         d = new BufferedReader(new InputStreamReader(fis));
562
563         //First, we store the file path of the .csv file
564         if (d.ready()){
565             filePath = nextValidLine(d);
566         }
567
568         //Next, we store if the csv file has headers or not
569         if (d.ready()){
570             hasHeaders = Boolean.parseBoolean(nextValidLine(d));
571         }
572
573         //Next, we store the number of input parameters
574         if (d.ready()){
575             numOfInput = Integer.valueOf(nextValidLine(d));
576         }
577
578         //Next, we store the number of output parameters
579         if (d.ready()){
580             numOfOutput = Integer.valueOf(nextValidLine(d));
581         }
582
583         //Next, we store the number of hidden layers
584         if (d.ready()){
585             numOfHiddenLayers = Integer.valueOf(nextValidLine(d));
586         }
587
588         //Next, we store the information for our hidden layers
589         allMyLayers = new LayerInfo[numOfHiddenLayers+2];
590
591         String layer = null;
592         int activationFunction;

```



```

593     boolean isBiased;
594     int neurons;
595
596     for (int i = 1; i < numOfHiddenLayers+1; i++){
597         if (d.ready()){
598             layer = nextValidLine(d);
599             layer = layer.trim().toLowerCase();
600             layer = layer.substring(1,layer.length()-1);
601             String[] layers = layer.split(",");
602
603             for (String l:layers){
604                 l = l.trim();
605             }
606
607             activationFunction = Integer.valueOf(layers[0].trim());
608             isBiased = Boolean.parseBoolean(layers[1].trim());
609             neurons = Integer.valueOf(layers[2].trim());
610
611             allMyLayers[i] =
612                 new LayerInfo(activationFunction, isBiased, neurons);
613         }
614     }
615
616     //Next, we store the information for the input layer
617     if (d.ready()){
618         layer = nextValidLine(d);
619         layer = layer.trim().toLowerCase();
620         layer = layer.substring(1,layer.length()-1);
621
622         isBiased = Boolean.parseBoolean(layer.trim());
623
624         allMyLayers[0] = new LayerInfo(-1, isBiased, numOfInput);
625     }
626
627     //Finally, we store the information for the output layer
628     if (d.ready()){
629         layer = nextValidLine(d);
630         layer = layer.trim().toLowerCase();
631         layer = layer.substring(1,layer.length()-1);
632
633         String[] layers = layer.split(",");
634
635         activationFunction = Integer.valueOf(layers[0].trim());
636
637         allMyLayers[numOfHiddenLayers+1] =
638             new LayerInfo(activationFunction, false, numOfOutput);

```

```

639     }
640
641     //store the information about the output 1 file type
642     if (d.ready()){
643         output1Type = Integer.valueOf(nextValidLine(d));
644     }
645
646     //store the information about the output 1 name
647     if (d.ready()){
648         output1Name = nextValidLine(d);
649     }
650
651     //store the information about the output 2 file type
652     if (d.ready()){
653         output2Type = Integer.valueOf(nextValidLine(d));
654     }
655
656     //store the information about the output 2 name
657     if (d.ready()){
658         output2Name = nextValidLine(d);
659     }
660
661     //Store the information for the desired training error
662     if (d.ready()){
663         desiredTrainingError = Double.valueOf(nextValidLine(d));
664     }
665
666     //Store the information for the maximum number of epochs
667     if (d.ready()){
668         numOfEpochs = Integer.valueOf(nextValidLine(d));
669     }
670
671     //Store the information for the desired network type
672     if (d.ready()){
673         networkType = Integer.valueOf(nextValidLine(d));
674     }
675
676     //We need additional variables if we are using Backpropagation
677     if (networkType == 1){
678         //Store the information for the learning rate
679         if (d.ready()){
680             learningRate = Double.valueOf(nextValidLine(d));
681         }
682
683         //Store the information for the momentum
684         if (d.ready()){

```

```

685         momentum = Double.valueOf(nextValidLine(d));
686     }
687 }
688
689 //TODO: reorder this
690 //output the information from the config file
691 System.out.println("config file validated:");
692 System.out.println("\tfilePath = " + filePath);
693 System.out.println("\thasHeaders = " + hasHeaders);
694 System.out.println("\tnumOfInput = " + numOfInput);
695 System.out.println("\tnumOfOutput = " + numOfOutput);
696 System.out.println("\tnumOfHiddenLayers = " + numOfHiddenLayers);
697 for (LayerInfo l: allMyLayers){
698     System.out.println("\t" + l.toString());
699 }
700 System.out.println("\tdesiredTrainingError = "
701     + desiredTrainingError);
702 System.out.println("\tnumOfEpochs = " + numOfEpochs);
703 System.out.println("\tnetworkType = " + networkType);
704 if (networkType == 1){
705     System.out.println("\tlearningRate = " + learningRate);
706     System.out.println("\tmomentum = " + momentum);
707 }
708 System.out.println("\toutput2Type = " + output1Type);
709 System.out.println("\toutput2Name = " + output1Name);
710 System.out.println("\toutput2Type = " + output2Type);
711 System.out.println("\toutput2Name = " + output2Name);
712
713 }
714 catch (Exception e){
715     //TODO: create more detailed error messages, to see where the error
716     //occurred
717     System.out.println("Invalid config file");
718     e.printStackTrace();
719 }
720 System.out.println("");
721 }
722 }

```

A.2 LayerInfo.java

```

1 package skynet;
2 /**
3  * A simple class, designed to hold the information required to create a
4  * layer in the neural network
5  * @author bwinrich
6  */

```

```

7
8 class LayerInfo{
9
10     /**
11      * An integer for the type of activation function (see comments in
12      * config file for details)
13      */
14     private int activationFunction;
15
16     /**
17      * A boolean for if the layer has a bias node or not
18      */
19     private boolean isBiased;
20
21     /**
22      * An integer for the number of normal neurons in the layer
23      */
24     private int neurons;
25
26     /**
27      * A constructor with parameters. We have no need for a default
28      * constructor
29      * @param activationFunction type of activation function
30      * @param isBiased is there a bias node
31      * @param neurons number of normal neurons
32      */
33     public LayerInfo(int activationFunction, boolean isBiased, int neurons){
34         this.activationFunction = activationFunction;
35         this.isBiased = isBiased;
36         this.neurons = neurons;
37     }
38
39     /**
40      * Accessor method for activationFunction
41      * @return the activationFunction
42      */
43     public int getActivationFunction() {
44         return activationFunction;
45     }
46
47     /**
48      * Accessor method for isBiased
49      * @return the isBiased
50      */
51     public boolean isBiased() {
52         return isBiased;

```

```

53     }
54
55     /**
56      * Accessor method for neurons
57      * @return the neurons
58      */
59     public int getNeurons() {
60         return neurons;
61     }
62
63     /** A method used for returning the information for the layer in an
64      * easy-to-read format, so that it can be printed.
65      * @see java.lang.Object#toString()
66      */
67     @Override
68     public String toString(){
69
70         String activation = null;
71
72         switch(activationFunction){
73             case -1:
74                 activation = "n/a";
75                 break;
76             case 0:
77                 activation = "Sigmoid";
78                 break;
79             case 1:
80                 activation = "Hyperbolic Tangent";
81                 break;
82             case 2:
83                 activation = "Linear";
84                 break;
85             case 3:
86                 activation = "Elliott";
87                 break;
88             case 4:
89                 activation = "Gaussian";
90                 break;
91             case 5:
92                 activation = "Logarithmic";
93                 break;
94             case 6:
95                 activation = "Ramp";
96                 break;
97             case 7:
98                 activation = "Sine";

```

```

99         break;
100     case 8:
101         activation = "Step";
102         break;
103     case 9:
104         activation = "BiPolar";
105         break;
106     case 10:
107         activation = "Bipolar Sigmoid";
108         break;
109     case 11:
110         activation = "Clipped Linear";
111         break;
112     case 12:
113         activation = "Competitive";
114         break;
115     case 13:
116         activation = "Elliott Symmetric";
117         break;
118     case 14:
119         activation = "Softmax";
120         break;
121     case 15:
122         activation = "Steepened Sigmoid";
123         break;
124     default:
125         activation = "Invalid";
126         break;
127     }
128
129     return ("Layer: (" + activation + ", " + isBiased + ", " + neurons
130           + ")");
131 }
132 }

```

A.3 OutputWriter.java

```

1  package skynet;
2  import java.io.BufferedWriter;
3  import java.io.File;
4  import java.io.IOException;
5  import org.encog.engine.network.activation.*;
6  import org.encog.neural.flat.FlatNetwork;
7  import org.encog.neural.networks.BasicNetwork;
8
9  /**
10   * A parent class for other OutputWriters. This class holds all of the

```

```

11  * shared methods required to create a file and output the code/formula for
12  * a trained Artificial Neural Network.
13  * @author bwinrich
14  */
15  public abstract class OutputWriter {
16
17      /**
18       * The file to write to
19       */
20      protected File file2;
21
22      /**
23       * A BufferedWriter for file writing
24       */
25      protected BufferedWriter bw2 = null;
26
27      /**
28       * The name of the file
29       */
30      protected String outputName;
31
32      /**
33       * Does the data set have headers?
34       */
35      protected boolean hasHeaders;
36
37      /**
38       * Array to hold the column names (only if the data set has headers)
39       */
40      protected String[] columnNames;
41
42      /**
43       * The data structure for the ANN
44       */
45      protected BasicNetwork network;
46
47      /**
48       * The number of input nodes
49       */
50      protected int inputCount;
51
52      /**
53       * The number of output nodes
54       */
55      protected int outputCount;
56

```

```

57  /**
58   * The number of neurons in each layer (including bias neurons)
59   */
60  protected int[] numberOfTotalNeurons;
61
62  /**
63   * The number of neurons in each layer (excluding bias neurons)
64   */
65  protected int[] numberOfNormalNeurons;
66
67  /**
68   * The value of the biases of each layer, if applicable
69   */
70  protected double[] biases;
71
72  /**
73   * The flattened version of the ANN
74   */
75  protected FlatNetwork myFlat;
76
77  /**
78   * The number of layers in the ANN
79   */
80  protected int layers;
81
82
83  /**
84   * Default constructor
85   */
86  public OutputWriter(){}
87
88  /**
89   * Mutator method for hasHeaders
90   * @param hasHeaders the hasHeaders to set
91   */
92  public void setHasHeaders(boolean hasHeaders) {
93      this.hasHeaders = hasHeaders;
94  }
95
96  /**
97   * Mutator method for columnNames
98   * @param columnNames the columnNames to set
99   */
100  public void setColumnNames(String[] columnNames) {
101      this.columnNames = columnNames;
102  }

```



```

103
104  /**
105   * Mutator method for network
106   * @param network the network to set
107   */
108  public void setNetwork(BasicNetwork network) {
109      this.network = network;
110  }
111
112  /**
113   * Mutator method for inputCount
114   * @param inputCount the inputCount to set
115   */
116  public void setInputCount(int inputCount) {
117      this.inputCount = inputCount;
118  }
119
120  /**
121   * Mutator method for outputCount
122   * @param outputCount the outputCount to set
123   */
124  public void setOutputCount(int outputCount) {
125      this.outputCount = outputCount;
126  }
127
128  /**
129   * Mutator method for numberOfTotalNeurons
130   * @param numberOfTotalNeurons the numberOfTotalNeurons to set
131   */
132  public void setNumberOfTotalNeurons(int[] numberOfTotalNeurons) {
133      this.numberOfTotalNeurons = numberOfTotalNeurons;
134  }
135
136  /**
137   * Mutator method for numberOfNormalNeurons
138   * @param numberOfNormalNeurons the numberOfNormalNeurons to set
139   */
140  public void setNumberOfNormalNeurons(int[] numberOfNormalNeurons) {
141      this.numberOfNormalNeurons = numberOfNormalNeurons;
142  }
143
144  /**
145   * Mutator method for biases
146   * @param biases the biases to set
147   */
148  private void setBiases(double[] biases) {

```

```

149         this.biases = biases;
150     }
151
152     /**
153      * Mutator method for myFlat
154      * @param myFlat the myFlat to set
155      */
156     private void setMyFlat(FlatNetwork myFlat) {
157         this.myFlat = myFlat;
158     }
159
160     /**
161      * Mutator method for layers
162      * @param layers the layers to set
163      */
164     public void setLayers(int layers) {
165         this.layers = layers;
166     }
167
168     /**
169      * Some variables can be initialized using information already passed to
170      * the class.
171      */
172     public void initializeOtherVariables(){
173         setMyFlat(network.getStructure().getFlat());
174         setBiases(myFlat.getBiasActivation());
175     }
176
177
178     /**
179      * Creates the file used for output.
180      * @param output2Name the name of
181      */
182     public abstract void createFile(String output2Name);
183
184     /**
185      * Writes to the output file. Each String passed as a parameter is
186      * written on its own line
187      * @param stuff The line to be written to the file
188      */
189     protected void writeTwo(String stuff){
190         try{
191             bw2.write(stuff);
192             bw2.newLine();
193         }catch (IOException e){
194             // TODO Auto-generated catch block

```

```

195         e.printStackTrace();
196     }
197 }
198
199 /**
200  * Parses the equation of the activation function and returns it in
201  * String form
202  * @param af The activation function to parse
203  * @param varName The variable passed to the activation function
204  * @param targetVarName The variable the result of the activation
205  * function will be stored in
206  * @return The parsed form of the activation function in String form
207  */
208 protected abstract String parseActivationFunction(ActivationFunction af,
209     String varName, String targetVarName);
210
211 /**
212  * A lengthy method for writing the code/formula for the neural network
213  * to a file. Each child class will have its own implementation.
214  */
215 public abstract void writeFile();
216 }

```

A.4 OutputWriterTxt.java

```

1 package skynet;
2 import java.io.BufferedWriter;
3 import java.io.File;
4 import java.io.FileWriter;
5 import java.io.IOException;
6 import org.encog.engine.network.activation.*;
7
8 /**
9  * A child class of OutputWriter, used for creating .txt files
10  * @author bwinrich
11  */
12 public class OutputWriterTxt extends OutputWriter{
13
14     /**
15      * Default Constructor
16      */
17     public OutputWriterTxt(){
18
19         /* (non-Javadoc)
20          * @see OutputWriter#writeFile()
21          */
22         @Override

```

```

23 public void writeFile() {
24
25     writeTwo("//Variable declarations");
26
27     //Variables from headers of csv file, if applicable
28     if(hasHeaders){
29         writeTwo("//Header Names");
30         for (String s: columnNames){
31             writeTwo(s);
32         }
33     }
34
35     //variables - input layer
36     writeTwo("//Input layer");
37     for (int i = 0; i<inputCount; i++)
38     {
39         writeTwo("i"+ i);
40     }
41     for (int i = inputCount; i<numberOfTotalNeurons[0]; i++)
42     {
43         writeTwo("i" + i + " = " + biases[biases.length-1]);
44     }
45
46     //variables - hidden layers
47     writeTwo("//Hidden layer(s)");
48     for (int i=1; i<layers-1; i++){
49         writeTwo("//Hidden Layer " + i);
50         for (int j = 0; j<numberOfNormalNeurons[i]; j++){
51             for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
52                 writeTwo("h" + i + "n" + j + "f" + k);
53             }
54             writeTwo("h" + i + "n" + j + "t");
55             writeTwo("h" + i + "n" + j);
56         }
57         for (int j = numberOfNormalNeurons[i]; j<numberOfTotalNeurons[i];
58             j++){
59             writeTwo("h" + i + "n" + j + " = " + biases[biases.length-i-1]);
60         }
61     }
62
63     //variables - output layer
64     writeTwo("//Output layer");
65     for (int i=0; i<outputCount; i++){
66         for (int j = 0; j < numberOfTotalNeurons[layers-2]; j++){
67             writeTwo("o" + i + "f" + j);
68         }

```

```

69         writeTwo("o" + i + "t");
70         writeTwo("o" + i);
71     }
72
73     writeTwo("");
74
75     double weight;
76     String sum = "";
77
78     //Some extra code if we have headers, to set the default input
79     //variables to the header variables
80     if(hasHeaders){
81         for (int i = 0; i < inputCount; i++){
82             writeTwo("i" + i + " = " + columnNames[i]);
83         }
84     }
85
86     //Hidden layers calculation
87     for (int i = 1; i<layers-1; i++){
88         for (int j = 0; j<numberOfNormalNeurons[i]; j++){
89             writeTwo("");
90
91             sum = "";
92
93             for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
94                 weight = network.getWeight(i-1,k,j);
95                 if (i == 1)
96                 {
97                     writeTwo("h" + i + "n" + j + "f" + k + " = i" + k + " * "
98                             + weight);
99                 }else{
100                     writeTwo("h" + i + "n" + j + "f" + k + " = h" + (i-1) + "n"
101                             + k + " * " + weight);
102                 }
103
104                 if (k == 0){
105                     sum = "h" + i + "n" + j + "f" + k;
106                 }else{
107                     sum += " + h" + i + "n" + j + "f" + k;
108                 }
109             }
110             writeTwo("h" + i + "n" + j + "t = " + sum);
111
112             String af = parseActivationFunction(network.getActivation(i),
113                     "h" + i + "n" + j + "t", "h" + i + "n" + j);
114             writeTwo(af.substring(0,af.length()-1));

```

```

115     }
116 }
117
118 //Output layer calculation
119 writeTwo("");
120
121 sum = "";
122
123 for (int i = 0; i<outputCount; i++){
124     for (int j = 0; j<numberOfTotalNeurons[layers-2]; j++){
125         weight = network.getWeight(layers-2,j,i);
126         writeTwo ("o" + i + "f" + j + " = h" + (layers-2) + "n" + j
127             + " * " + weight);
128
129         if (j == 0){
130             sum = "o" + i + "f" + j;
131         }else{
132             sum += " + o" + i + "f" + j;
133         }
134     }
135     writeTwo("o" + i + "t = " + sum);
136
137     String af = parseActivationFunction(network.getActivation(layers-1),
138         "o" + i + "t", "o" + i);
139     writeTwo(af.substring(0,af.length()-1));
140 }
141
142 //Some extra code if we have headers, to set the default input
143 //variables to the header variables
144 if (hasHeaders){
145     writeTwo("");
146
147     for (int i = 0; i < outputCount; i++){
148         writeTwo(columnNames[i + inputCount] + " = o" + i);
149     }
150 }
151
152 writeTwo("");
153
154 try{
155     if(bw2!=null)
156         bw2.close();
157 }catch(Exception ex){
158     System.out.println("Error in closing the BufferedWriter"+ex);
159 }
160 }

```

```

161
162  /* (non-Javadoc)
163   * @see OutputWriter#createFile(java.lang.String)
164   */
165  @Override
166  public void createFile(String output2Name){
167
168      outputName = output2Name;
169
170      try{
171          file2 = new File(output2Name + ".txt");
172          if (!file2.exists()) {
173              file2.createNewFile();
174          }
175
176          FileWriter fw2 = new FileWriter(file2);
177          bw2 = new BufferedWriter(fw2);
178      }catch (IOException e){
179          // TODO Auto-generated catch block
180          e.printStackTrace();
181      }
182  }
183
184  @Override
185  protected String parseActivationFunction(ActivationFunction af,
186      String varName, String targetVarName){
187      String text = null;
188
189      if (af instanceof ActivationSigmoid){
190          text = targetVarName + " = 1.0 / (1.0 + e^(-1 * " + varName + "))";
191      }else if (af instanceof ActivationTANH){
192          text = targetVarName + " = tanh(" + varName + ")";
193      }else if (af instanceof ActivationLinear){
194          text = targetVarName + " = " + varName;
195      }else if (af instanceof ActivationElliott){
196          double s = af.getParams()[0];
197          text = targetVarName + " = ((" + varName + " * " + s
198              + ") / 2) / (1 + |" + varName + " * " + s + "|) + 0.5";
199      }else if (af instanceof ActivationGaussian){
200          text = targetVarName + " = e^(-(2.5*" + varName + ")^2)";
201      }else if (af instanceof ActivationLOG){
202          text = "if(" + varName + " >= 0){\n\t" + targetVarName
203              + " = log(1 + " + varName + ")\n}else{\n\t" + targetVarName
204              + " = -log(1 - " + varName + ")\n}";
205      }else if (af instanceof ActivationRamp){
206          double paramRampHighThreshold =

```

```

207         ((ActivationRamp)(af)).getThresholdHigh();
208     double paramRampLowThreshold =
209         ((ActivationRamp)(af)).getThresholdLow();
210     double paramRampHigh = ((ActivationRamp)(af)).getHigh();
211     double paramRampLow = ((ActivationRamp)(af)).getLow();
212     double slope = (paramRampHighThreshold-paramRampLowThreshold)
213         / (paramRampHigh-paramRampLow);
214
215     text = "if(" + varName + " < " + paramRampLowThreshold + ") {\n\t"
216         + targetVarName + " = " + paramRampLow + "\n} else if ("
217         + varName + " > " + paramRampHighThreshold + ") {\n\t"
218         + targetVarName + " = " + paramRampHigh + "\n} else {\n\t"
219         + targetVarName + " = (" + slope + " * " + varName + ")";
220 }else if (af instanceof ActivationSIN){
221     text = targetVarName + " = sin(2.0*" + varName + ")";
222 }else if (af instanceof ActivationStep){
223     double paramStepCenter = ((ActivationStep)(af)).getCenter();
224     double paramStepLow = ((ActivationStep)(af)).getLow();
225     double paramStepHigh = ((ActivationStep)(af)).getHigh();
226
227     text = "if (" + varName + ">= " + paramStepCenter + ") {\n\t"
228         + targetVarName + " = " + paramStepHigh + "\n} else {\n\t"
229         + targetVarName + " = " + paramStepLow + "\n}";
230 }else if (af instanceof ActivationBiPolar){
231     text = "if(" + varName + " > 0) {\n\t" + targetVarName
232         + " = 1\n} else {\n\t" + targetVarName + " = -1\n}";
233 }else if (af instanceof ActivationBipolarSteepenedSigmoid){
234     text = targetVarName + " = (2.0 / (1.0 + e^(-4.9 * " + varName
235         + "))) - 1.0";
236 }else if (af instanceof ActivationClippedLinear){
237     text = "if(" + varName + " < -1.0) {\n\t" + targetVarName
238         + " = -1.0\n} else if (" + varName + " > 1.0) {\n\t"
239         + targetVarName + " = 1.0\n} else {\n\t" + targetVarName
240         + " = " + varName + "\n}";
241 }else if (af instanceof ActivationElliottSymmetric){
242     double s = af.getParams()[0];
243     text = targetVarName + " = (" + varName + "*" + s + ") / (1 + |"
244         + varName + "*" + s + "|)";
245 }else if (af instanceof ActivationSteepenedSigmoid){
246     text = targetVarName + " = 1.0 / (1.0 + e^(-4.9 * " + varName
247         + ")))";
248 }else{
249     //Unimplemented activation function: Softmax (complicated)
250     //Unimplemented activation function: Competitive (complicated,
251     //non-differentiable)
252     //in Encog 3.3 there aren't any other activation functions, so

```



```

253         //unless someone implements their own we shouldn't get to this point
254         text = "Error: unknown activation function";
255     }
256     return text;
257 }
258 }

```

A.5 OutputWriterJava.java

```

1  package skynet;
2  import java.io.BufferedWriter;
3  import java.io.File;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import org.encog.engine.network.activation.*;
7
8  /**
9   * A child class out OutputWriter, used for creating .java files.
10  * @author bwinrich
11  */
12  public class OutputWriterJava extends OutputWriter{
13
14      private boolean standalone;
15
16      /**
17       * Default constructor
18       */
19      public OutputWriterJava(){}
20
21      /**
22       * Constructor with parameter
23       * @param standalone main method or not
24       */
25      public OutputWriterJava(boolean standalone){
26          this.standalone=standalone;
27      }
28
29      /* (non-Javadoc)
30       * @see OutputWriter#writeFile()
31       */
32      @Override
33      public void writeFile() {
34
35          writeTwo("import java.lang.Math;");
36
37          writeTwo("");
38

```

```

39 writeTwo("public class " + outputName);
40 writeTwo("{}");
41
42 writeTwo("\t//Variable declarations");
43
44 //Variables from headers of csv file, if applicable
45 if(hasHeaders){
46     writeTwo("\t//Header Names");
47     for (String s: columnNames){
48         writeTwo("\tpublic static double " + s + ";");
49     }
50 }
51
52 //variables - input layer
53 writeTwo("\t//Input layer");
54 for (int i = 0; i<inputCount; i++)
55 {
56     writeTwo("\tpublic static double i"+ i + ";");
57 }
58 for (int i = inputCount; i<numberOfTotalNeurons[0]; i++)
59 {
60     writeTwo("\tprivate static double i" + i + " = "
61         + biases[biases.length-1] + ";");
62 }
63
64 //variables - hidden layers
65 writeTwo("\t//Hidden layer(s)");
66 for (int i=1; i<layers-1; i++){
67     writeTwo("\t//Hidden Layer " + i);
68     for (int j = 0; j<numberOfNormalNeurons[i]; j++){
69         for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
70             writeTwo("\tprivate static double h" + i + "n" + j + "f" + k
71                 + ";");
72         }
73         writeTwo("\tprivate static double h" + i + "n" + j + "t;");
74         writeTwo("\tprivate static double h" + i + "n" + j + ";");
75     }
76     for (int j = numberOfNormalNeurons[i]; j<numberOfTotalNeurons[i];
77         j++){
78         writeTwo("\tprivate static double h" + i + "n" + j + " = "
79             + biases[biases.length-i-1]+";");
80     }
81 }
82
83 //variables - output layer
84 writeTwo("\t//Output layer");

```

```

85     for (int i=0; i<outputCount; i++){
86         for (int j = 0; j < numberOfTotalNeurons[layers-2]; j++){
87             writeTwo("\tprivate static double o" + i + "f" + j + ";");
88         }
89         writeTwo("\tprivate static double o" + i + "t;");
90         writeTwo("\tpublic static double o" + i + ";");
91     }
92
93     writeTwo("");
94
95     //standalone files will have a main method
96     if(standalone){
97
98         //TODO: customize this (from Tiberius)
99         writeTwo("\tpublic static void main(String[] args)");
100        writeTwo("\t{");
101        writeTwo("\t\tinitData();");
102        writeTwo("\t\tcalcNet();");
103        for (int i = 0; i < outputCount; i++){
104            writeTwo("\t\tSystem.out.println(o" + i + ");" );
105        }
106        writeTwo("\t}");
107        writeTwo("");
108
109        //TODO: customize this (from Tiberius)
110        writeTwo("\tpublic static void initData()");
111        writeTwo("\t{");
112        writeTwo("\t\t//data is set here");
113        for (int i = 0; i < inputCount; i++){
114            if(hasHeaders){
115                writeTwo("\t\t" + columnNames[i] + " = 1;");
116            }else{
117                writeTwo("\t\ti" + i + " = 1;");
118            }
119        }
120        writeTwo("\t}");
121        writeTwo("");
122    }
123
124    double weight;
125    String sum = "";
126
127    writeTwo("\tpublic static void calcNet()");
128    writeTwo("\t{");
129
130    //Some extra code if we have headers, to set the default input

```

```

131 //variables to the header variables
132 if (hasHeaders){
133     for (int i = 0; i < inputCount; i++){
134         writeTwo("\t\ti" + i + " = " + columnNames[i] + ";");
135     }
136 }
137
138 //Hidden layers calculation
139 for (int i = 1; i<layers-1; i++){
140     for (int j = 0; j<numberOfNormalNeurons[i]; j++){
141         writeTwo("");
142
143         sum = "";
144
145         for (int k = 0; k < numberOfTotalNeurons[i-1]; k++){
146             weight = network.getWeight(i-1,k,j);
147             if (i == 1)
148             {
149                 writeTwo("\t\tth" + i + "n" + j + "f" + k + " = i" + k
150                     + " * " + weight + ";");
151             }else{
152                 writeTwo("\t\tth" + i + "n" + j + "f" + k + " = h" + (i-1)
153                     + "n" + k + " * " + weight + ";");
154             }
155
156             if (k == 0){
157                 sum = "h" + i + "n" + j + "f" + k;
158             }else{
159                 sum += " + h" + i + "n" + j + "f" + k;
160             }
161         }
162
163         writeTwo("\t\tth" + i + "n" + j + "t = " + sum + ";");
164
165         String af = parseActivationFunction(network.getActivation(i),
166             "h" + i + "n" + j + "t", "h" + i + "n" + j);
167         writeTwo("\t\t" + af);
168     }
169 }
170
171 //Output layer calculation
172 writeTwo("");
173
174 sum = "";
175
176 for (int i = 0; i<outputCount; i++){

```

```

177         for (int j = 0; j<numberOfTotalNeurons[layers-2]; j++){
178             weight = network.getWeight(layers-2,j,i);
179             writeTwo("\t\t" + i + "f" + j + " = h" + (layers-2) + "n" + j
180                 + " * " + weight + ";");
181
182             if (j == 0){
183                 sum = "o" + i + "f" + j;
184             }else{
185                 sum += " + o" + i + "f" + j;
186             }
187         }
188         writeTwo("\t\t" + i + "t = " + sum + ";");
189
190         String af = parseActivationFunction(network.getActivation(layers-1),
191             "o" + i + "t", "o" + i);
192         writeTwo("\t\t" + af);
193     }
194
195     //Some extra code if we have headers, to set the default input
196     //variables to the header variables
197     if (hasHeaders){
198         writeTwo("");
199         for (int i = 0; i < outputCount; i++){
200             writeTwo("\t\t" + columnNames[i + inputCount] + " = o" + i
201                 + ";");
202         }
203     }
204
205     writeTwo("\t}");
206
207     writeTwo("");
208     writeTwo("}");
209
210     try{
211         if(bw2!=null)
212             bw2.close();
213     }catch(Exception ex){
214         System.out.println("Error in closing the BufferedWriter"+ex);
215     }
216 }
217
218 /* (non-Javadoc)
219  * @see OutputWriter#createFile(java.lang.String)
220  */
221 @Override
222 public void createFile(String output2Name){

```

```

223
224     outputName = output2Name;
225
226     try{
227         file2 = new File(output2Name + ".java");
228         if (!file2.exists()) {
229             file2.createNewFile();
230         }
231
232         FileWriter fw2 = new FileWriter(file2);
233         bw2 = new BufferedWriter(fw2);
234     }catch (IOException e){
235         // TODO Auto-generated catch block
236         e.printStackTrace();
237     }
238 }
239
240 @Override
241 protected String parseActivationFunction(ActivationFunction af,
242     String varName, String targetVarName){
243     String text = null;
244
245     if (af instanceof ActivationSigmoid){
246         text = targetVarName + " = 1.0 / (1.0 + Math.exp(-1 * " + varName
247             + "));";
248     }else if (af instanceof ActivationTANH){
249         text = targetVarName + " = Math.tanh(" + varName + "));";
250     }else if (af instanceof ActivationLinear){
251         text = targetVarName + " = " + varName + ";";
252     }else if (af instanceof ActivationElliott){
253         double s = af.getParams()[0];
254         text = targetVarName + " = ((" + varName + " * " + s
255             + ") / 2) / (1 + Math.abs(" + varName + " * " + s
256             + ")) + 0.5;";
257     }else if (af instanceof ActivationGaussian){
258         text = targetVarName + " = Math.exp(-Math.pow(2.5*" + varName
259             + ", 2.0));";
260     }else if (af instanceof ActivationLOG){
261         text = "if(" + varName + " >= 0){\n\t" + targetVarName
262             + " = Math.log(1 + " + varName + ");\n}else{\n\t"
263             + targetVarName + " = -Math.log(1 - " + varName + ");\n}";
264     }else if (af instanceof ActivationRamp){
265         double paramRampHighThreshold =
266             ((ActivationRamp)(af)).getThresholdHigh();
267         double paramRampLowThreshold =
268             ((ActivationRamp)(af)).getThresholdLow();

```

```

269     double paramRampHigh = ((ActivationRamp)(af)).getHigh();
270     double paramRampLow = ((ActivationRamp)(af)).getLow();
271     double slope = (paramRampHighThreshold-paramRampLowThreshold)
272         / (paramRampHigh-paramRampLow);
273
274     text = "if(" + varName + " < " + paramRampLowThreshold + ") {\n\t"
275         + targetVarName + " = " + paramRampLow + ";\n} else if ("
276         + varName + " > " + paramRampHighThreshold + ") {\n\t"
277         + targetVarName + " = " + paramRampHigh + ";\n} else {\n\t"
278         + targetVarName + " = (" + slope + " * " + varName + ");";
279 }else if (af instanceof ActivationSIN){
280     text = targetVarName + " = Math.sin(2.0*" + varName + ");";
281 }else if (af instanceof ActivationStep){
282     double paramStepCenter = ((ActivationStep)(af)).getCenter();
283     double paramStepLow = ((ActivationStep)(af)).getLow();
284     double paramStepHigh = ((ActivationStep)(af)).getHigh();
285
286     text = "if (" + varName + ">= " + paramStepCenter + ") {\n\t"
287         + targetVarName + " = " + paramStepHigh + ";\n} else {\n\t"
288         + targetVarName + " = " + paramStepLow + ";\n}";
289 }else if (af instanceof ActivationBiPolar){
290     text = "if(" + varName + " > 0) {\n\t" + targetVarName
291         + " = 1;\n} else {\n\t" + targetVarName + " = -1;\n}";
292 }else if (af instanceof ActivationBipolarSteepenedSigmoid){
293     text = targetVarName + " = (2.0 / (1.0 + Math.exp(-4.9 * "
294         + varName + "))) - 1.0;";
295 }else if (af instanceof ActivationClippedLinear){
296
297     text = "if(" + varName + " < -1.0) {\n\t" + targetVarName
298         + " = -1.0;\n} else if (" + varName + " > 1.0) {\n\t"
299         + targetVarName + " = 1.0;\n} else {\n\t" + targetVarName
300         + " = " + varName + ";\n}";
301 }else if (af instanceof ActivationElliottSymmetric){
302     double s = af.getParams()[0];
303     text = targetVarName + " = (" + varName + "*" + s
304         + ") / (1 + Math.abs(" + varName + "*" + s + "));";
305 }else if (af instanceof ActivationSteepenedSigmoid){
306     text = targetVarName + " = 1.0 / (1.0 + Math.exp(-4.9 * " + varName
307         + "));";
308 }else{
309     //Unimplemented activation function: Softmax (complicated)
310     //Unimplemented activation function: Competitive (complicated,
311     //non-differentiable)
312     //in Encog 3.3 there aren't any other activation functions, so
313     //unless someone implements their own we shouldn't get to this point
314     text = "Error: unknown activation function";

```

```

315     }
316     return text;
317 }
318 }

```

A.6 TrainingSetUtil.java (modified)

```

1  /*
2   * Encog(tm) Core v3.3 - Java Version
3   * http://www.heatonresearch.com/encog/
4   * https://github.com/encog/encog-java-core
5
6   * Copyright 2008-2014 Heaton Research, Inc.
7   *
8   * Licensed under the Apache License, Version 2.0 (the "License");
9   * you may not use this file except in compliance with the License.
10  * You may obtain a copy of the License at
11  *
12  *     http://www.apache.org/licenses/LICENSE-2.0
13  *
14  * Unless required by applicable law or agreed to in writing, software
15  * distributed under the License is distributed on an "AS IS" BASIS,
16  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
17  * See the License for the specific language governing permissions and
18  * limitations under the License.
19  *
20  * For more information on Heaton Research copyrights, licenses
21  * and trademarks visit:
22  * http://www.heatonresearch.com/copyright
23  */
24  package org.encog.util.simple;
25
26  /*
27   * modified: added condition so that it will ignore rows with incomplete
28   * data
29   */
30  /*
31   * Additional modification: added way to retrieve header information
32   */
33
34  import java.util.ArrayList;
35  import java.util.List;
36
37  import org.encog.ml.data.MLData;
38  import org.encog.ml.data.MLDataPair;
39  import org.encog.ml.data.MLDataSet;

```



```

40 import org.encog.ml.data.basic.BasicMLData;
41 import org.encog.ml.data.basic.BasicMLDataPair;
42 import org.encog.ml.data.basic.BasicMLDataSet;
43 import org.encog.util.EngineArray;
44 import org.encog.util.ObjectPair;
45 import org.encog.util.csv.CSVError;
46 import org.encog.util.csv.CSVFormat;
47 import org.encog.util.csv.ReadCSV;
48
49 public class TrainingSetUtil {
50
51     private static List<String> columnNames = new ArrayList<String>();
52
53     /**
54      * Load a CSV file into a memory dataset.
55      * @param format The CSV format to use.
56      * @param filename The filename to load.
57      * @param headers True if there is a header line.
58      * @param inputSize The input size. Input always comes first in a file.
59      * @param idealSize The ideal size, 0 for unsupervised.
60      * @return A NeuralDataSet that holds the contents of the CSV file.
61      */
62     public static MLDataSet loadCSVTOMemory(CSVFormat format,
63         String filename, boolean headers, int inputSize, int idealSize) {
64         MLDataSet result = new BasicMLDataSet();
65         ReadCSV csv = new ReadCSV(filename, headers, format);
66
67         if(headers){
68             columnNames = csv.getColumnNames();
69         }
70
71
72         int ignored = 0;
73
74         while (csv.next()) {
75             MLData input = null;
76             MLData ideal = null;
77             int index = 0;
78             try{
79                 input = new BasicMLData(inputSize);
80                 for (int i = 0; i < inputSize; i++) {
81                     double d = csv.getDouble(index++);
82                     input.setData(i, d);
83                 }
84
85                 if (idealSize > 0) {

```

```

86         ideal = new BasicMLData(idealSize);
87         for (int i = 0; i < idealSize; i++) {
88             double d = csv.getDouble(index++);
89             ideal.setData(i, d);
90         }
91     }
92
93     MLDataPair pair = new BasicMLDataPair(input, ideal);
94     result.add(pair);
95 }catch (CSVError e){
96     ignored++;
97
98     //e.printStackTrace();
99 }
100 }
101 System.out.println("Rows ignored: " + ignored);
102
103 return result;
104 }
105
106 public static ObjectPair<double[][]> trainingToArray(
107     MLDataSet training) {
108     int length = (int)training.getRecordCount();
109     double[][] a = new double[length][training.getInputSize()];
110     double[][] b = new double[length][training.getIdealSize()];
111
112     int index = 0;
113     for (MLDataPair pair : training) {
114         EngineArray.arrayCopy(pair.getInputArray(), a[index]);
115         EngineArray.arrayCopy(pair.getIdealArray(), b[index]);
116         index++;
117     }
118
119     return new ObjectPair<double[][]>(a, b);
120 }
121
122 /**
123  * @return the columnNames
124  */
125 public static List<String> getColumnNames() {
126     return columnNames;
127 }
128
129 }
130 }

```

BIBLIOGRAPHY

- Charles, D. and Mcglinchey, S., “The past, present and future of artificial neural networks in digital games,” *Proceedings of the 5th international conference on computer games: artificial intelligence, design and education*, pp. 163–169, 2004.
- Clabaugh, C., Myszewski, D., and Pang, J. “History: The 1940’s to the 1970’s.” [Online]. Available: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html>
- Cortes, C. and Vapnik, V., “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: <http://dx.doi.org/10.1007/BF00994018>
- Elliott, D. L., “A better activation function for artificial neural networks,” 1993.
- Galway, L., Charles, D., and Black, M., “Machine learning in digital games: a survey,” *Artificial Intelligence Review*, vol. 29, no. 2, pp. 123–161, 2008.
- Heaton, J. “The number of hidden layers.” September 2008. [Online]. Available: <http://www.heatonresearch.com/node/707>
- Heaton, J. “Elliott activation function.” September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Elliott_Activation_Function
- Heaton, J. “Range normalization.” September 2011. [Online]. Available: http://www.heatonresearch.com/wiki/Range_Normalization
- Hebb, D., “The organization of behavior; a neuropsychological theory.” 1949.
- Jain, A. K., Mao, J., and Mohiuddin, K., “Artificial neural networks: A tutorial,” 1996.
- McCulloch, W. and Pitts, W., “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943. [Online]. Available: <http://dx.doi.org/10.1007/BF02478259>
- Riedmiller, M. and Braun, H., “A direct adaptive method for faster backpropagation learning: The rprop algorithm,” in *Neural Networks, 1993., IEEE International Conference on.* IEEE, 1993, pp. 586–591.
- Rojas, R., “The backpropagation algorithm,” in *Neural Networks.* Springer, 1996, pp. 149–182.

- Rosenblatt, F., “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*, vol. 65, no. 6, p. 386, 1958.
- Stuart, K. “How to get into the games industry – an insiders’ guide.” March 2014. [Online]. Available: <http://www.theguardian.com/technology/2014/mar/20/how-to-get-into-the-games-industry-an-insiders-guide>
- Widrow, B., Hoff, M. E., *et al.*, “Adaptive switching circuits.” 1960.